Library
Naval Postgraduate School
Monterey, California 93940

A GENERAL STRUCTURE FOR UNCOOPERATIVE PROCESSES DISTRIBUTED OVER A SYSTEM NETWORK

BY

JOHN FREDERICK KRAMER, JR.

A thesis submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN

1973

Library
Naval Postgraduate School
Monterey, California 93940

ACKNOWLEDGEMENTS

I wish to gratefully acknowledge the United States Navy for permitting me to complete my education and for financial assistance during my study at the University of Wisconsin.

I would especially like to thank Professor D. R. Fitzwater for his ideas, continual support, encouragement, and helpful criticism; George Cowan and Vatche Papazian for helpful and stimulating discussions; John Hine for the initial development of the concepts of control and control-points; and finally the numerous other graduate students with whom I have taken courses, all of whom have contributed to the development of this thesis and have been ardent critics of my ideas. I would like to express my gratitude to Professors Desautels and Rees for their suggestions and editorial assistance which substantially improved the presentation of the ideas of this thesis, and to Marcella Helmus and Jean Cich for their hours of typing.

Finally, I am very deeply indebted to my wife, Bev, for having unswerving confidence and faith in my abilities and for her encouragement and devotion.

TABLE OF CONTENTS

ACKNOWLEDGMENTS. ii

TABLE OF CONTENTS. iii

LIST OF ILLUSTRATIONS. v

CHAPTER 1: SYSTEM STRUCTURES

Introduction. 1
Current Work. 3
Problem Statement 28

CHAPTER 2: NONCOOPERATION - THE SOLUTION

Introduction. 31
Postulated Design Constraints 35
Delegation of Authority Constraints 49
Implementation Designer Constraints 59
Process Designer Constraints. 60
Network Designer Constraints. 70

CHAPTER 3: NETWORK STRUCTURE DESIGN

Introduction. 77
Subprocessor and Expression-state 80
Control-Point 91
Communication 102
Definition, Modification, and Bounds. 109
Initial System Choices. 116

CHAPTER 4: SYSTEM COMPARISONS

Summary of Network Structures 126
Use as a Model. 135
Further Work 154
Summary and Conclusions 158

TABLE OF CONTENTS (Continued)

APPENDIX A: CONSTRAINTS 163

APPENDIX B: DESIGN DECISIONS. 167

APPENDIX C: EXAMPLE FORMAL NETWORK DEFINITION

Introduction. 170
Description of Network Definitions. 176
Formal Definition 188

BIBLIOGRAPHY 201

LIST OF ILLUSTRATIONS

FIGURE		Page
1	TENEX Model 1	137
2	TENEX Model 2	137
3	TENEX Model 3	137
4a	INTERPRETER Flowchart 1	173
4b	INTERPRETER Flowchart 2	174
4c	INTERPRETER Flowchart 3	175
5	SYSTEM PHASES	183

> The languages people use to communicate with computers differ in their intended aptitudes, towards either a particular application area, or a particular phase of computer use (high level programming, program assembly, job scheduling, etc.). They also differ in physical appearance, and more important in logical structure. The question arises, do the idiosyncrasies reflect basic logical properties of the situations that are being catered for? (La66, p. 164).

CHAPTER 1

SYSTEM STRUCTURES

Introduction

The last decade has produced many areas of interest within the field of computer science. This thesis is concerned with the subarea of computer systems, and more specifically the virtual design (abstracted from any particular physical hardware) for networks of asynchronously operating digital computers. In addition to being appropriate for networks, the general design presented in this thesis can be used to describe current logical operating systems and can provide a framework within which to design languages.

Early digital computers were very simple, had only limited structures within which users could formulate solutions to their problems, interpreted only simple

sequences of instructions, and were under the complete control of one user at a time. Input and output was normally under user program control, and only minimal control functions were provided such as test and skip on certain flags, limited jumps, and instruction modification.

Users soon desired to construct more complex computations. More general control structures were introduced along with ways of communicating between program parts. Extended machine languages were developed to exploit these new features. Users could now do more, and they soon found that such tasks as the routine testing for external interactions, were wasting much of their capacity. The resulting concept of interrupt, which was "apparently introduced with the UNIVAC 1103" to obtain good input/output (BN71) did away with this annoyance but introduced a more complicated control structure. A well-defined program organization was required so the users could cause the processor to focus its attention on the appropriate program part at the appropriate time, along with methods of resolving logical conflicts due to the asynchronous arrivals of these interrupts.

The need to improve system performance and lower job costs by using parallelism, in conjunction with the interrupt, resulted in multiprocessor computer systems. The use of simple I/O processors caused the CPU to be

poorly utilized when only one user program was resident in the machine at a time. Better CPU usage was achieved with multiprogramming operating systems (time multiplexed programs proceeding in a pseudo parallel manner). The concept of digital "process" as a sequence of process states was developed as the abstraction of these interacting entities. Early processes were not very complex, and although each process could proceed in pseudo parallel with other processes, little or no explicit interprocess interactions were allowed. As users became more sophisticated, more complex intraprocess state structures and control operations were developed. The process state normally includes both information to direct the processor in its operations and the information which the processor manipulates in response to such directions.

The ability to create multiple processes soon led to a desire to use a set of parallel processes to accomplish a single purpose thus requiring the support of more general interprocess interactions. New problems were created by multiprocess systems because of the need to share resources. The physical system needed to multiplex its physical assets to achieve some optimal goal, while the logical processes needed to share logical resources (for example, the information in files) amongst themselves while protecting these objects against misuse by a borrower.

As experimentation with interprocess and system user relationships proceeded, a need developed for the management of uncooperative processes. If a user process is completely isolated from all other processes including the system, as was true in earlier operating systems, it can be uncooperative and only the user of that process is affected. Such isolation is no longer feasible because of the diverse user population which must now be served. Many systems being developed are attempting to fulfill these needs by permitting a user to construct multiprocess computations and allowing substantial interprocess interactions including the control of other processes.

The design of a computer system to support a general user community can make no simplifying assumptions concerning their reliability. It must allow for users who are subject to making errors and who may even intentionally try to circumvent the system to achieve their own goals. One of the essential concerns when designing a general purpose system is that it be able to support such potentially uncooperative processes while guaranteeing the integrity of the system. As users employ more complex interprocess interactions, they require access to more complex facilities.

The delegation of such services and decisions to the user process opens the way for the possibility of

significant uncooperative behavior which the system can no longer prevent. In order to prevent such behavior, the system designer would have to make arbitrary a priori decisions as to what is to be considered uncooperative behavior by all of the processes to run on that system. However, if interprocess interactions are permitted, only the users of the system (through their processes) can decide, in general, when uncooperative behavior has occurred with respect to these interactions. It is impossible for the system to prevent uncooperative process behavior without imposing worse case restrictions on all processes.

The real need is to control the effects of such uncooperative behavior by regulating the scope of the effects of the interprocess interactions of each process. If one process within a system claims another is being unfair, either some outside authority must arbitrate or they must come to a joint solution, in which case they have implicitly formed a joint authority. A generalization of the system/user relationship is necessary if a network design is to permit delegation of the control of interesting interprocess interactions. Users must be delegated the authority to control the processes over which they have responsibility. For example, a user process should have the same ability to control the processes it creates, as a time sharing system process has to control the processes

it has created for users. Users must be provided a mechanism for exercising their authority, where the rules of responsibility say it is appropriate, thus permitting them to control the interactions of those processes which they feel need be controlled. At the same time though, those processes not under their responsibility, must be provided protection from their authority.

One final need becoming apparent today is for system designs to be portable. F. C. Withington (Wi72) makes some interesting observations about future computer systems. He claims that because there will be significant hardware cost performance improvements in the near future, machine independent applications which are portable will be more economical than "optimized" machine dependent applications. Application systems would not need to be rewritten solely because the physical hardware upon which they are implemented is changed. He predicts that the next generation will include network systems and then forecasts that

> the operating system will permit the network of processing modules to be widely varied; the user will be able to add, subtract, and upgrade them individually, without undergoing the pain of conversion. The 'generation cycle' we have become familiar with should end, and orderly evolution should become the rule.

Today's software systems are very complex and cannot be rewritten easily for a new hardware system. If

Withington's predictions are correct, the benefits of writing everything for a virtual (not machine dependent) system structure, and then implementing this structure on new physical devices would far outweigh the inefficiencies due to any specific implementation of that virtual system. The implementer of such a virtual system could use translation, or Waite's macros (Wa70), to ease the implementation job, but the user would not have to be concerned with the move. If the virtual system is augmentable, to meet special needs, users may be able to gain some of the same efficiencies they now get by writing for a particular physical machine while still getting portability.

In order for portability to be successful, a clear distinction must always be made between the virtual structures defined and the physical implementation of them. The general acceptance of virtual address spaces and virtual processors has proven that it is not necessary to require the logical and physical mapping to be 1-1. A general design, that clearly separates the logical and physical systems, is necessary if portability is to be achieved. Only then does the design not become constrained to a particular type of machine (perhaps at the cost of sacrificing some specific efficiencies of each machine).

With the advent of networks, composed of different

kinds of digital computers, transportability becomes even more imperative. If the complex is to be a useful vehicle for sharing resources, then the whole network must provide a consistent face to the user. Even if all of the communication interfaces are the same, the users will be reluctant to exploit the network if they must interface with a different operating system at each node. A user may wish to move a process between nodes. This means a standard interface must be defined for user processes, at a level which assume no particular physical architecture or devices. The virtual system designer should not define what particular physical resources the implementer must have. The implementers of the virtual system must be given the flexibility to select those physical resources which permit them to optimize their implementation subject only to local management constraints. If hardware is developed which will improve their implementation, they should be able to use it without requiring any change in the logical system the user sees. The only effect on the user should be a change in the cost performance.

Current Work

Since it is impossible to discuss all of the literature relevant to this thesis, attention will be focused primarily on those systems which represent the logical structure within which users define their problems and

not physical implementation or application problems. Since the COSINE report (Co71) contains a general discussion of the area of operating systems and provides an introduction to many of the terms that will be used in this review of current work, no real effort will be made to define specific terms here. Those terms which are used in later chapters will be explained and defined at that time within the context of the network being developed.

Of primary interest in this thesis are those works which deal with networks of digital computers, the process state structures supported by systems, and the operating environment provided for user processes. This includes the concepts of control and parallelism, the interactions among processes, and the ability of a user process to manage resources. Papers dealing with the protection and allocation of logical resources and with portable systems are also of interest. The remainder of this section will attempt to survey some of these areas to indicate the logical structures currently being used.

Networks: Networks of general purpose digital computers are becoming accepted as a viable way to provide computer services, and interest has developed in how to exploit such networks to allow increased availability of physical and logical resources. Farber (Fa72) defines a computer network as "an interconnected set of dependent

or independent computer systems which communicate with each other in order to share certain resources such as programs or data and/or for load sharing and reliability reasons."

The April 1972 Datamation from which the above quote was taken provides a brief introduction to seven "typical networks" and some of the problems involved including organization and management of such networks. Farber's article discusses the networks with respect to organization, composition, number and geography of nodes, machine sizes, and communication interfaces. Companion articles by Stefferud (St72) and Hootman (Ho72) look at networks from a manager's point of view, including such questions as "who is management," "where it resides," and network economics. None of the articles discuss what operating system structures must be provided in a network to fulfill these responsibilities, and none of them make a clear distinction between the physical systems and the logical systems which will be using the network. Implementation management problems relating to the accounting of physical assets is included, but not how the logical processes will be able to manage logical resources distributed over the network. The desire of users to create multiprocess computations and manage logical resources by user defined techniques is being overlooked, even though

it appears this may be desired more in a network than in current systems. Although the technical interconnection problems "show promise of being solved" (St72), the problems of how to use networks, and the political and management problems, must also be solved before a user will be able to write interesting and useful general systems which can exploit networks.

Information concerning the "ARPA" network is the most readily available. Two series of papers (CC70,HK70, RW70) and (CH72,OH72,TH72), along with some companion papers, provide a fairly complete discussion of it. The purpose of ARPANET was to provide "a network that would allow for the interconnection, via common-carrier circuits, of dissimilar computers at widely separated, ARPA-sponsored research centers" (OH72).

Each node has an IMP (Interface Message Processor) which "acts as a nodal switching unit and also interconnects the research computer centers" (HOSTS) (OH72). Originally, a user accessed the network only through a HOST, but it soon became desirable to provide direct terminal access which was done using a Terminal IMP (TIP). The TIP provides the services of connect and disconnect to remote sites, and then becomes transparent to the conversation between user and HOST. It is not the intention of this thesis to go into the implementation problems of

HOST to HOST communication. The interested reader is referred to the papers for a more complete description of these problems.

"The growth of the network has dynamically catalyzed an area of computer science which has to do with the quite general problem of how programs should intercommunicate, whether in a single computer or between computers" (OH72). In the ARPANET, an attempt was made to provide a general mechanism allowing user-level process-to-process communication (CH72). Interprocess communication is carried out in a simplex manner using one of the 256 logical connections. Since each HOST in the network names its own processes, "sockets" provide a unique name in a name space divided among the HOSTs. Each HOST then maps socket names onto the processes. Processes are created according to procedures local to their respective HOSTs, and must interface with a network control program (NCP) resident in each HOST to be able to communicate. If the conversation between processes is to be duplex, two connections must be made.

A process uses local operating system calls to tell the NCP to send a message to a specific socket. The NCP interacts with the ARPANET to send the message to the receiver's HOST NCP, which then uses appropriate system primitives to deliver the message to the process with the

destination socket. One major weakness of the ARPANET is that "System calls will vary from one operating system to another" (CC70). The ARPANET has no concept of a network operating system it provides for its processes.

Process: The use of the term "process," as a concept relevant to computer systems became common in the literature as a result of its use by two separate groups: project MAC at MIT used it in 1964 as a "precise concept in discussing design problems of multiprogrammed computer systems" (De70) and Dijkstra used it in 1965 when he published his work on "Cooperating Sequential Processes" (Di67). It is somewhat surprising that the connotations of the many formal and informal definitions of process are similar even though each definition was designed to fit within a certain context. A survey of these definitions can be found in the introduction to Horning and Randall's "Process Structuring" (HR71). Their definition of process is as follows:

> A triple (S, f, s), where S is a state variable set, f is an action function on that set, and s is the subset of the state space of S which defines the initial states of a process. A process generates all the computations generated by its action function from its initial states.

Fitzwater and Hintz (FH71) present a definitional system which allows the designer to represent a process state, define a processor, and then validate the design

by observing the system behavior as the processor works on that state. Their definition of process is an "interpretation sequence of process states." This definitional system is useful because it relates processes to the concepts of system, complexes of such systems, and asynchronous communication between the members of this complex. This system is convenient for providing a formal definition of a design to remove any ambiguities.

A digital computer system has the property that there is one or more points in its execution sequence at which its state can be observed and its behavior studied. A process state normally contains a process name space or data part, and a program part which directs a processor in its transformation on the process state.

Although the ARPA papers talk about creating a process in a HOST system and allowing processes in separate HOSTs to communicate using the network, there is no formalism of the properties of a process or of its states. Each system is controlled by the physical system manager and there are no network operations, other than communication, by which one process can interact with a process in some other system.

Although identifiable processes, as components in computer systems, were originally very simple, they have become more general and more complex. Dijkstra

(Di65,67,68,71) has investigated logical process cooperation, when no restrictions exist on their relative speeds. The cooperating processes have access to one or more variables, which they can cooperatively check to see whether or not to proceed into a "critical section." Although a process may have to wait logically on other processes before entering its critical section, it proceeds asynchronously at all other times.

Numerous recent designs use the concept of a process in describing their system. "THE" (Di68) is a "society of sequential processes with undefined speed ratios" where "harmonious cooperation is regulated by means of explicit synchronizing statements." Alsberg's (Al71) "OSL/2" is an implementation of Dijkstra's hierarchial concepts in which a process can create and monitor new processes, thus becoming an operating system for them. "ESOPE" (BB69) permits processes to interact via interprocess communication and control other processes by using activate, block, create and destroy. "TENEX" (BB71) permits a user to create a process tree hierarchy with multiple simultaneously runnable processes where a parent controls the children. The "RC4000" system (Ha71a) is basically a "monitor that can be extended with a hierarchy of operating systems." Any program can initiate "other programs in a hierarchial manner and execute them

according to any strategy desired." In all of these systems, a process is normally considered as a potentially asynchronous executable entity with communication sometimes possible between it and other processes in the system.

Control: Some concept of control (what is to be done next) is present in one form or another in all systems and computer languages. When talking about programs Dennis (DV66) describes a "locus of control" and Denning (De68) a "thread of control." With respect to multiprogramming and multiprocessing systems, control may be considered as the switching of the attention of the processors among the various programs in the system, and can be described at various levels of abstraction (for example, machine instruction execution, program instruction sequencing, or multiprogramming).

Fisher (Fi70) says "the control structure of a programming language is its sequencing or interpretation rules and as such provides a programming environment which encompasses any task within that language." He goes on to say that there is a need for more than a "step by step" concept of control so the "point of view" with which problems can be attacked is not constrained. Both his introduction and that of Herriot (He71) provide good discussions of control.

Control structures in most early computer languages were not much more complex than the machines upon which they were designed to be run. One instruction was sequentially executed after another with conditionals, jumps, and later subroutine jumps as their only control mechanisms. Systems were normally dedicated to one user at a time, and it was only to provide services for the convenience of the users that a basic operating system was developed. Although operating system designers soon included concepts of system/user relationships, multiprogramming, and multiprocessing, there was little development of languages containing such concepts.

SIMULA (DN67) introduced the concept of coroutine and SIMULA 67 (DM68) extended this to "quasi-parallel systems." Nested blocks similar to SIMULA main blocks were allowed, resulting in a tree of nested coroutines. "Taskmaster" (FM65a, FM65b) (a real time system) similarly organizes its tasks in a tree structure. Every Taskmaster task has several ALGOL like statements which may receive control directly from the scheduler, or as a result of clock or outside interrupts, rather than sequentially from other tasks. A significant disadvantage of "Taskmaster" is the requirement that all users cooperatively return control to the scheduler before the next task can be executed.

Dijkstra's work with cooperating sequential processes has interested researchers in the definition and implementation of systems permitting process interactions, along with the associated problems of mutual exclusion and synchronization. The implementation of "P&V" type operations usually is based on having only one physical processor or a sharable memory (OSL/2 (Al71), ESOPE (BB69), OREGANO (BE71), THE (Di68), Venus (Li71)). Efficient implementation of "P&V" operations, when these assumptions are not true, might be difficult. Other systems permit interprocess interactions through a form of logical communications other than the use of shared variables (GE600 (BD69), TENEX (BB71), ARPA). One example of how the solution of "critical section" problems using "P&V" operations could be difficult to implement (on a multiprocessor system), would be when the processors are using a "cache" like buffer memory. There is normally no update of the information buffered by one processor if another processor changes a buffered variable. Special features would have to be introduced just to handle the "P&V" operations. The ARPANET was forced to use a message system instead of "P&V" type operations for interprocess communications since processes could reside in different HOSTs which did not share memories. Wirth (Wi69) complains that "P&V" may be too primitive anyway, since the synchronization of

loosely connected processes naturally consists of the interchange of messages, as is done in ordinary life.

Most systems do not guarantee either the order of execution or the length of time a physical processor will be applied to each process. Individual processes effectively proceed asynchronously, except through their own explicit logical synchronization. For example, a common way of handling interprocess communication is by synchronizing the two processes through such primitives as block/wakeup or block/cause (BD69, SO69). Requiring a process to go to sleep, when waiting for a message, implies a single thread of control in a process and no logical interrupt facilities. Wirth (Wi69) feels this is good because "The interrupt effectively consists of the insertion of one or several instructions at points of the program which can neither be foreseen nor reconstructed." On the other hand, it prevents the process from doing other work while waiting for messages. OSL/2 (Al71) has no concept of an interrupt at the logical level, and as the author points out, must be completely "command driven." Rose (Ro71) is the only author who specifically discusses "multi-threaded" processes, although multiple threads are implicit in the few systems which do allow transfer to a specific address for the delivery of a message.

System-programmers recognized early the usefulness

of faults (error conditions which occur during the execution of an operator in a processor), and interrupts (signals from sources other than the processor itself) but user application languages have been slow to include such concepts. PL/I (IB68) introduced the "ON condition," but most other languages have no such feature. Needham (Ne71) feels that all faults should be handled within the process using a trapping mechanism. As a general mechanism, he says a process should have a "catastrophic address" to which control is transferred when all else fails. The major problem with most present implementations of faults is that they are not handled by the user in the environment in which they occurred, but are treated as interrupts to independent environments. This distinction only becomes essential in multiprocess systems. The system designer cannot expect to know everything each user wants to do, and therefore can implement only arbitrary design decisions in such system provided fault handling routines.

Interactions and Resources: Early computer systems were not concerned with interprocess interactions since they supported only one process (user) at a time. As systems became more complex, the number of concurrent users increased, and the amount of resident information accessible to the executing processes increased. Uncooperative process behavior by design or default became a dangerous

potentiality, and the control of interprocess interactions became a necessity. At first the trend was to isolate the process completely, both logically and physically, but it soon became apparent that suitable facilities had to be provided to permit processes to interact.

Since interprocess interactions (mutual or reciprocal action or influence of one process on another) will always exist, some form of protection must be provided. A good overview of process protection can be found in Co71 and De71. Most computer systems, designed for multiprogramming provide some control over process interactions, such as privileged instructions or memory protection. For example, the IBM 360 memory hardware protect and the Multics (Or72) segmentation binding within a ring, were designed to limit a process's access to its logical address space regardless of where it resided physically. Use of a hierarchical directory, with the system checking access rights, is the usual way of protecting user files. One problem in Venus (Li71) is the lack of segment protection which requires but cannot ensure that all processes be cooperative. Most of the other systems make use of disjoint logical process address space (GE600 (BD69), ESOPE (BB19)). TENEX (BB71) and Multics (Or72) integrate the file system into the virtual process address space, binding segments to a logical name only at execution time. They hope to

make sharing of data and procedures simpler since two logical names can be used by different processes, and can be mapped onto one physical name representing the shared information. One of the major difficulties of permitting interprocess interactions only through shared segments is that the occurrence of race conditions is encouraged, which must be solved by expensive cooperation, and requires physical systems to always provide a way of sharing memories even in a network. Interprocess interactions using communication do not have this difficulty.

Achieving process protection by using separate logical address spaces, created a need for sharing logical entities. The concept of a logical resource which can be passed to a process was developed. Which entities are to be considered resources, and which processes are to be permitted to have them, is highly dependent on the level of abstraction with which the system is being managed. Until recently resources were defined primarily in terms of their mapping onto an implementation. The introduction of multiprogramming, and the concepts of virtual machines and virtual address spaces, have begun to separate what the process manipulates from the resources the implementer manipulates. Pnysical resources such as space are manipulated by the implementor, and logical resources such as files are manipulated by the logical processes.

Some systems are beginning to permit processes to manipulate resources. In "Venus" (Li71) cooperative resource management is used. In "Multics" (Or72), programs and information can be shared freely at the will of the user. In "ESOPE" (BB69) a set of resources is allocated to each user, and it is up to the set of user processes to share them. Gaines (Ga71) limits a child to a subset of its parents' resources, as does the RC 4000 (Ha71a) system.

With the advent of suballocation of resources, theoretical work on the protection of logical objects becomes much more important. Vanderbilt (Va69) controls access to shared items by executing certain objects within a data structure language he has developed. Lampson (La69, 71) is interested both in the theories of protection and in implementation problems such as "how can information which specifies protection and authorizes access, itself be protected and manipulated?" His theory suggests that objects be named by capabilities, access be granted to these objects by keys. A process then can be considered to execute in a domain (group of capabilities), and entry between domains can be protected by "gates" thus giving a process different powers in different domains. Graham (Gr71a) extends this work, and discusses the problem in relation to "Sue." Conway (CM72) suggests another

implementation model where the matrix elements "are decision rules." He makes a distinction between data dependent and data independent rules which can be exploited by checking the data independent security once at translation time and then checking only the data dependent security at execution time.

The protection of objects is imperative, and any general logical system must provide some technique for doing this. A decision must be made as to which objects are to be protected, the means of protecting each, and the rights of the owners of such protected objects. A design cannot be considered as properly supporting uncooperative processes unless protection is provided for all interactions.

Cooperative Management of Uncooperative Processes: Most systems today lack a general mechanism, which the process can invoke, for handling uncooperative processes, particularly when the process to be controlled may exist in some other system in a network. To be able to do this, mechanisms must be provided to allow for accountable resource allocation and for control (possibly remote) of interprocess interactions. Recently there have been several logical system designs which have partial solutions to some of these problems, but they are too specialized to satisfy all of the requirements (processes which

must share address space to interact, cannot be completely isolated).

The definition of uncooperative behavior between interacting processes is only possible from the point of view of the authority over such processes. Several designers have included process hierarchies in their systems which permits a process to control its own subprocesses.

> Since the difference between operating systems and production programs is one of jurisdiction only, this problem is solved by arranging the internal processes in a hierarchy in which parent processes have complete control over child processes. (Ha70)

A child can only be created, started, stopped, and removed by a parent who supplies the resources and gets them back when the child is removed.

Gaines (Ga71) is interested in "what happens if a process derails," and TENEX (BB71) has an "inverted tree defined by the capability for direct control and killing." "Although not completely general, a tree structure process hierarchy implicitly provides the protection and reference facilities that are wanted in most applications." One of the design goals of TENEX was that "the system should encourage and facilitate cooperation among users as well as provide protection against undesirable interactions." There are three areas which might be considered deficiencies if this system was to be considered as a network structure. First, processes are permitted to communicate

via shared memory, which is difficult if the hierarchy is distributed over nodes of the network. Second, each "Job" is a separate hierarchy, thus limiting some of the general interprocess cooperation which might be desirable. Third, the system does not provide for logical resources to be allocated down a hierarchy with guaranteed protection.

Portability: If useful portability is to be possible, each system must maintain a strict separation of logical and physical resources, and the logical system that a user sees should remain invariant over moves to different physical systems. Landin (La66) says "The physical/logical terminology is often used to distinguish features that are a fortuitous consequence of physical conditions from features that are in some sense more essential." Given any specific logical structure to implement, the implementer usually has many ways to do it. The fact that the physical system is effectively finite usually requires certain constraints in the logical processes. For example, "THE" (Di68) requires all processes sharing system resources to state, before the request is made, what their maximum needs for each type will be.

Although, within the concept of a virtual processor, users cannot directly control many of the physical resources, virtual processors as currently used do not solve the problems of portability because the virtual processor usually

has an extremely close relationship to the physical processor on which it is implemented. Recently several systems have been implemented using a layered hierarchy of systems or "onion skin" structure. Portability is then achieved by designing a logical structure at the user's interface and then implementing this structure as a layered hierarchy of more abstract virtual machines with the outermost layer being the logical system the user sees. Project LOGOS uses this structure (Br72, Gl72a, b, HR72, RB72). "The concept of a layered hierarchy of machines is consistent with some current trends in system architecture and applies to both hardware and software of a system. Thus the hardware/software split can be postponed in the design process, allowing greater flexibility of implementation." Several systems have used this technique in their implementations ("THE" and "Venus"). The design of a logical network which is abstracted from any particular machine permits different network facilities to implement the design using any desired number of hierarchies of systems. Using this technique, the implementer may find that it is only the lowest levels of the hierarchy which need redefining. Regardless of what the facility implementer must do to provide the system, the user will see the same basic logical structure when the system is finally implemented.

Problem Statement

Any general but useful design for networks of digital computers must solve the problems of network manageability, generality, and portability. The network designer must determine which decisions should be built into the design, and which should be delegated to the users or the implementers to make at a later time. The more decisions built into a network design, the less adaptable the network will become and the less freedom the user will have to create optimal process relationships. A solution, which does not result in arbitrary or unnecessary constraints on the network users or implementers, is to design into the system itself, only the choices necessary to permit the network designer to delegate all other decisions to either the users or the implementers.

The system designed in this thesis, provides sufficient factorization of responsibilities and delegation of authority to control the behavior of the processes running in the network, to permit the development of a network which satisfies the following postulated constraints.

1. The designed network must consist of a set of interacting bounded programmable digital computer systems with well-defined processes, and be open to communication with foreign systems having undefined processes.

2. Responsibility for meeting the postulated

design constraints and the design decisions must be factored between the network, implementation, and process designers.

3. Each designer must have sufficient authority to control the effects, on the processes running in the network, of the interactions for which that designer has been delegated responsibility.

4. Although processes must be permitted to delegate to other processes their authority to control process interactions, the delegating process always retains responsibility for that interaction and the authority to control the process to which it was delegated.

5. The network definition must provide for modification of its definition. The design must provide sufficient constraints to be able to delegate safely to process designers all modification decisions and authority to control the effects of such decisions on the processes in the network.

The postulated constraints must always be satisfied when design decisions are made in this thesis. Those design decisions influenced by the postulated constraints must be shown to provide the greatest generality consistent with the postulated and derived constraints. Those decisions which are not directly influenced by the constraints need only be made subject to good, but informal,

design concepts.

Chapter 2 of this thesis contains the development of a set of constraints within which the network decicions will be made. The postulated constraints are motivated first, and then additional constraints, which follow directly from them, are developed. The third chapter develops and informally defines the network design consistent with the constraints developed in Chapter 2. Chapter 4 uses the network structures to describe several other representative systems, discusses both the implications of this work, and of other work being done in parallel with this thesis, and describes some of the areas which deserve future research. A formal definition of the designed network is given in Appendix C.

> Once a satisfactory set of criteria and postulates has been formulated, then one may deduce, rather than design, the structure of a computer (Van Horn, Va66, p. 227).

CHAPTER 2

NONCOOPERATION - THE SOLUTION

Introduction

The previous chapter described some of the problem areas associated with current networks and systems, along with some of the present approaches to solving those problems. This chapter will develop a set of design constraints sufficient to allow the solution of network management problems arising from the inclusion of general control structures, potentially uncooperative processes, and portability of user processes and the design itself. The problem of process noncooperation will be looked at briefly, and the postulated constraints will be motivated. Additional constraints, which define the delegation of authority to control certain process interactions, will be developed from the postulated constraints.

A solution to the problem of process noncooperation is necessary if a network designer is to be able to give process designers responsibility and authority to control the processes in the network. A direct approach to the

problem of process noncooperation is difficult because there is no objective point of view from which to decide that noncooperation exists. What is considered cooperative by one process designer, might be considered uncooperative by another. A final judgment, by definition, can only be made by the proper authority, which might exist as a single designer or cooperatively as joint designers. Three levels of authority exist in most computer systems today: user, operating system, and implementation. One of the major problems, usually encountered in current systems, is the lack of a clear factorization, between these three groups, of the responsibility for the control of process behavior and authority to do so. Operating system and implementation designers (commonly indistinguishable) often know very little about what the user intended a process to do, and therefore can only make general assumptions about it when trying to manage its behavior.

It is unlikely that the network designer will ever know all viewpoints from which to judge process noncooperation since this would require complete knowledge of what all users of the network were logically trying to accomplish. Analysis, by the network designer, of process noncooperation would only lead to nonformal systems and a philosophical discussion. The network designer should not make arbitrary decisions for the user by decreeing which

process actions are to be considered cooperative and which are not, but should provide services to the appropriate agent responsible for that process to make the decision.

The key to the problem of process noncooperation is that there must be at least two interacting parties before cooperation or lack of it can have any meaning. It takes at least two marching soldiers before they can be out of step. The one declared out of step is clearly not the ranking authority unless it was a cooperative self declaration because of incompetence. If all interactions in the network can be recognized, and a safe means provided by which they can be controlled, the network designer can delegate responsibility and authority for the management of such interactions to other agents. A logical state structure, as distinct from a physical implementation of it, must be imposed on the processes along with the definition of appropriate transformations on these states, so that the effects of all process operations are known. If the scope of the effects of a processes' potential noncooperation is well-defined, responsibility for process noncooperation can be delegated to the processes concerned. (Well-defined: Something is considered well defined if it has been formally defined and the definition holds over all systems in the network.)

If process designers are to be delegated the

responsibility for controlling process noncooperation, mechanisms must be provided within the network design for the control of process interactions through which the effects of noncooperation can be transmitted. As systems became more complex, more than one dimension of responsibility and authority for the control of processes developed. Users of early computers had complete control of the machine and hence were completely responsible for their processes, but as systems became more complex, the user was no longer uniquely responsible for what was happening.

For each interaction between processes, a proper partitioning would designate a responsible agent who could then be given authority to control that interaction. Interactions between processes can occur because of the existence of a process, the logical expectations of a process, and interprocess communication.

The existence of a process may cause interactions with other processes because the implementation must use some of its limited physical resources to represent it. The birth of a new process or death of an old one can possibly cause interactions because of limited rights to create or kill a process, the use of objects borrowed from other processes, or the use of some of the physical processer time during the operation. Interactions might also occur if a process is able to manipulate the physical sys-

tem to get faster service or more resources, generate invalid state structures, or to change a microprogram which another process might need to use.

Interactions due to the logical expectations of one process with respect to the behavior of another process make up a second category of potential sources of noncooperation. Scheduling expectations might include not doing what another process wants, doing things in the wrong order, or not giving control back. A processes' management of resources might cause problems by not returning resources, allocating them improperly, not accounting for what has been done with them, and changing or destroying them.

Finally, interprocess communication can be a potential source of noncooperation. A process might not be able to find out the status of another process if the other process is disregarding inputs or not sending outputs. Messages might be garbled or lost if input processing is too slow. Shared address spaces might permit improper state changes, modifying or destroying commonly addressed objects, as well as both logical and physical race problems with respect to accessing common objects.

Although the above examples are not complete, they do indicate some of the interactions which a network design must consider. The network designers and implementers

can be responsible for controlling some of them, however the responsibility for the majority of them must be delegated to the process designers since in many situations only the user has the knowledge required to recognize uncooperative processes.

Postulated Design Constraints

There are five postulated design constraints which are sufficient to permit a network to be designed which can be used to provide a solution to network management problems. Each constraint will have a discussion following it which is intended to explain the terms used in it and motivate why the constraint was included.

> Constraint C1 - The designed network must consist of a set of interacting bounded programmable digital computer systems with well-defined processes, and be open to communication with foreign systems having undefined processes.

Bounded - A logical system will be considered to be bounded if it defines the logical effects of implementation failures which may occur because the implementer has only a limited amount of physical resources of limited reliability with which to work. The logical system does not define when the implementation failures will occur; this can only be determined by an evaluation of each implementation and may in fact change with time.

Well-defined processes - A process will be considered well-defined if it has been formally defined and if the definition holds over all systems in the network.

Foreign system - A system that interacts with the systems in the network and which is constrained by the network designers only in the nature of these interactions.

Undefined process - Any process which is not formally defined in the network. We can therefore make no assumptions about its behavior.

The first constraint is intended to limit the scope of the network structures which will be developed in the next chapter. It should always be kept in mind that this is a general logical network design and as such makes no assumptions about the physical systems upon which it may be implemented. With the advent of networks, any general system design must consider the possibility that the processes residing in different systems may wish to interact.

A design which is appropriate for a network of interconnected computer systems can be constrained to run on a single system, but it may not be possible to extend a single system design to run on a network and still provide satisfactory generality. The situation where the network provides little more than interconnection services

between systems, and each logical operating system is developed separately, is also not satisfactory. The user, whose processes span several systems in the network, must design each process to operate properly on its HOST system which may limit the computation's flexibility (TH72). This problem can be avoided if the design includes a network as a basic postulate and then includes process state structures and operations on these states which are as general as possible. By providing a common logical system design at each system in the network, the design permits the user to easily understand the network and the processes it will support. It is a network design that is of interest in this thesis.

"Interacting"--When looking at an arbitrary set of systems, it is convenient to be able to say whether or not each member of the set is also a member of the network. Only systems which interact with other systems of the network can be candidates for membership in the network. If a system does not interact with a system in the network then it is not of interest in this thesis. Foreign systems are not considered to be members of the network.

"Bounded" systems must be included in the design in recognition of the fact that the physical implementation will, for all practical purposes, have access to only limited physical resources. The network design must

recognize this fact, and properly designate responsibility for the process interactions which can occur due to it.

"Programmable"--If the design were not restricted to programmable computer systems, several networks could be designed which might not be considered general. Systems which exhibit totally built in reflexes (not programmable by the user), such as airline reservation systems, although interesting, have a design which could not be easily extended to a network where the users are defining many of the processes. On the other hand, a design providing for user programmable systems, could be used to implement the reflex system.

"Digital computer" systems have several properties which make them nice to study. The design problems they present should be solved before looking at more complex networks. Although the formal definition of "digital system" being used in this thesis can be found in FH71 and Jo73, the essential concepts will be introduced here informally. It is important to recognize that there is a distinction between a single digital system and a network of interacting digital systems. A digital system has three primary parts: a set of components which hold information; a set of devices which are capable of performing transformations on the information stored in those components; and a set of communication "channels."

The "system processor" is that part of the digital system which performs the transformations and is itself invariant with time. It may contain one or more operators which carry out transformations on process states. Each operator is applied as follows. All operators which are to be executed during the present system cycle are applied in parallel at the beginning of the system cycle and then proceed asynchronously. The next system cycle does not commence until all operators have completed and the system has again reached a defined state.

A "system process" is an initial instantaneous description of the whole state of the system, followed by all of those states reached by repetitive application of the system processor. The system state may contain a set of one or more process states which represent different parts of the system state (for example, user process states). A "process" within a digital system can thus be defined by its initial process state and the sequence of process states reached as the system processor operators are applied to it.

Each digital system in a network has a system state and system processor which perform the transformations on that state. Each system processor proceeds at its own pace through its interpretation sequence, without any consideration of the speed at which the other members of the

set are proceeding. Because there is no synchronization between different systems, there must be a clear understanding of intersystem communication. Each system proceeds as follows. If messages are generated during the application of the system processor, they are collected in a set, and when the system processor completes its cycle they are transmitted to the other systems. After transmitting the messages, the system updates its own state with the messages that have arrived since the previous update and the next cycle begins. Incoming messages are buffered between updates since (in a digital system) they must not influence the interpretation during the application of the system processor. This is closely analogous to the ordinary concepts of intersystem interrupts.

"Well-defined" processes are necessary before any assumptions can be made concerning their behavior. For example, the creation of a new user process state must be controlled so it can be recognized as such by the system processor. Each process state may have several parts which must remain invariant, irrespective of what is happening in that system or in any other system in the network.

A network will be considered "open" if the processes, when executing on one of the network systems, have

the ability to communicate messages to and receive messages from foreign systems (using normal interprocess communication only). Although it would have been possible to have considered a closed network, where communication could be exchanged only between processes in the network, there would have been many interesting process interactions which would have been ruled out. Since it would be difficult for the network designer to be able to know how to interpret the logical structure of all future message interactions with foreign systems, this is left up to the process to do.

The same mechanism will be used for communication with foreign systems as is used for interprocess messages between processes in the network. As a result, no special structures will have to be provided by the network designer, to service these external interactions. Even though foreign systems will not be formally defined in the network and therefore nothing can be said about the behavior of their processes, as long as the communication operators are formally defined, and appropriate constraints placed on their effects on the network systems, the network processes can still remain formally defined. Since the network designers have no authority over the design of foreign systems, they must localize the potential effects of process interactions with such systems. By restricting such interactions to messages to and from a process in the

network, potential interactions are limited to that process. The network design must only ensure that such message interactions are possible, and that the processes have control over the receipt and interpretation of such messages.

> Constraint C2 - Responsibility for meeting the postulated design constraints and design decisions must be factored between the network, implementation, and process designers.

In this thesis it is only those interactions, for which one of these three designers may be held responsible, that will be discussed. Although other designers (managements) may exist in a network and may be important, they are not of interest here. In order for the network to be manageable, there must be no conflicts of authority or improper assignment of responsibility. For these three designers there must be a partitioning of responsibility which is disjoint in time. The network designer influences process interactions by the features included in the design, the implementation designer influences process behavior by the physical resources chosen with which the design is implemented, and the process designer is responsible for controlling process behavior by the features chosen with which to define it.

The separation of logical and physical resources

found in some systems today is part of such factorization. The physical resources are managed by the implementation. Logical processes make requests to use certain resources, but the mapping from logical to physical is not necessarily 1 - 1. Since not all implementations in a network will come under one management, the network design provides the common interface between the implementers and the process designers. If responsibility is clearly factored, each designer can solve its own management problems in a locally optimal manner. The development of more complex systems and the distribution of a set of processes over a network make it imperative that the responsibilities of the network, implementation, and process designers be clearly stated. Identification of these responsibilities will permit the network designer to introduce appropriate mechanisms so each responsible agent can fulfill its responsibilities.

Of primary interest here are the process state structures and system processor operators necessary for the process and network designers to do their job. Although the implementation of the network design on particular architectures will require much design work, no assumptions will be made here about the way the physical implementers will solve their problems. Until there is a network design to implement, questions concerning im-

plementation are only secondary.

Constraint C2 requires that responsibility for meeting the postulated design constraints and design decisions, including the interactions mentioned earlier, be assigned to one of the three designers. Within the network, each implementation management must be free to manage the resources it is using to implement the network. Therefore it must have responsibility for such resources so it can meet its own optimization criteria. If users are to be permitted to construct interesting and useful process relationships, process designers must be assigned responsibility for the definition of these processes. The network design must, of course, define the interfaces between each of the other designs, and therefore must have responsibility for providing a well-defined interface and for controlling interactions between the other designs.

> Constraint C3 - Each designer must have sufficient authority to control the effects, on the processes running in the network, of the interactions for which that designer has been delegated responsibility.

If responsibility for controlling the effects of process interactions is factored between network, implementation, and process designers, then each designer must define and control uncooperative process interactions. Since noncooperation is a point of view, only the re-

sponsible agent can determine when some action is uncooperative. It was pointed out earlier that process noncooperation can be controlled only by restricting the effects of the various interactions through which it can occur. Each of the three designers, therefore, must be given such authority to restrict the effects of the interactions for which it is responsible. Only then can each designer implement a local control of cooperation or lack of it.

Although constraint C2 requires that responsibility be factored between the three designers, nothing would have been solved without the addition of constraint C3. Responsibility to control process interactions without the associated authority to carry out those responsibilities would be meaningless. The network design must include sufficient logical process state structures and system processor operators for a designated responsible agent to control those interactions which the design has defined as its responsibilities. Responsibility and authority must both reside with the same agent.

> Constraint C4 - Although processes must be permitted to delegate to other processes their authority to control process interactions, the delegating process always retains responsibility for that interaction and the authority to control the process to which it was delegated.

This constraint must be postulated if process de-

signers are to have the flexibility to do their job. The intent is not to design a process independent system, but just the opposite. The design should permit the user to define multiprocess computations which are highly process dependent. Several recent system designs include such a service to the users (TENEX, RC4000).

The question of whether or not a given process designer should make use of multiprocess computations does not belong here, although it is not difficult to think of examples where doing so makes design of process behavior much simpler. A simple example would be a logical operating system which has three primary processes running under it (time sharing, batch, and utility). The operating system designer might want to have one process, which retains ultimate authority, to arbitrate conflicts between the other three processes, but to delegate the authority to make local decisions to the three children.

The delivery of maximum freedom to user processes to manage other local processes, requires that they be considered as submanagers and given corresponding authority to control process behavior necessary to fulfill their responsibilities. Process submanagers should be able to construct their computations as multiple interacting processes and be able to delegate some of their authority to subprocesses over which they retain ultimate responsibility

and authority. The user can then create various operating environments for sets of subprocesses in a hierarchy of user defined operating systems. Instead of requiring one process to make all decisions according to a general algorithm, authority can be passed to other processes to make the decisions in a locally optimal manner. Constraint C4 is intended to force the network designer to include the ability for processes to delegate responsibility and authority. A clear designation will have to be made as to which responsibilities the network designer will delegate to the process designers.

> Constraint C5 - The network definition must provide for modification of its definition. The design must provide sufficient constraints to be able to delegate safely to process designers all modification decisions and authority to control the effects of such decisions on the processes.

Any network design, which is to survive over a period of time, must recognize the fact that changes will be necessary. Additional systems may be added to the network or new process state structures may develop. Constraint C5 requires the final network design to define what types of network evolutions will be permitted, along with the environments in which they may take place and under what constraints. If these evolutionary processes were not so constrained, an implementation designer could

never know that the implementation was valid.

Process designer support is the whole reason for any computer system design. It would be inconsistent to give process designers responsibilities for some interaction and then permit the network to be modified and effect that interaction without the "consent" of that designer. If the network is to support well-defined processes as required by constraint C1, process designers must make the decisions as to whether or not some modification will cause some of their processes to misbehave. Implicit in this constraint is the fact that, because of the three way factoring of responsibility for controlling process interactions, there will be some interactions which will be the responsibility of the network and implementation designers and therefore should not be available to the process designers. The control that process designer has over the effects of network definition modifications will be determined by the factoring of responsibility required by constraint C2.

Delegation of Authority Constraints

The previous section introduced the design constraints that are being postulated in this thesis. The remainder of this chapter will develop some additional constraints which follow from the postulated constraints.

This section looks at some authority delegation constraints arising primarily from C2 and C3.

> Delegation Constraint D1 - Responsibility for each interaction and authority to control it must be uniquely assigned to the same designer.

This constraint follows directly from C2 and C3. Although it is implicit in the previous constraints, it should be clearly stated as a constraint so the network designer makes such a unique designation when a choice exists. If a designer is designated as responsible for controlling an interaction, it would not be consistent to permit another agent to also have control over it. The responsible agent would no longer be responsible and constraint C2 would be violated. A designer with authority to control an interaction must be able to act on its own initiative, subject only to its design constraints.

A constraint similar to D1 is necessary in any design that is to produce a manageable system. Each designer must be able to solve its problems in any locally optimal manner, without fear that some other designer is also competitively controlling that same interaction with resulting conflict of authorities. If conflicts of authority existed in a system, the system would no longer be guaranteed to be manageable unless there were an arbitrator present to settle the conflict. If such an authority does

exist, then it is the unique authority this constraint says must exist.

The recognition that this constraint exists, provides the network designer the freedom to delegate to the other designers the responsibility for solving their own problems. Once an interaction is designated as being the responsibility of one of the three groups of designers, mechanisms can be provided to permit those designers to solve their problems. The network design does not have to make arbitrary choices of algorithms to solve particular problems, but can defer the choice of which algorithm to use to the appropriate designer. Unique assignment of authority permits the designer to delegate the maximum number of responsibilities to the facility and process designers. This results in a simplified network design which still fulfills the responsibilities of the network designers.

> Delegation Constraint D2 - Network designers are responsible for ensuring that the network is self-protecting, meets the constraints, and is formally defined.

Constraint D2 follows directly from the second postulated constraint requiring that responsibility for interactions be assigned to one of three designers. The network design provides the interface between the process and implementation managements. The remainder of this

thesis will define this interface.

Self-protecting - A network is self-protecting if there are sufficient constraints on the freedoms of future designers (implementation and process) so their actions cannot affect the network in any way which would cause it to violate any of the constraints or design decisions.

If the design were not made self-protecting, constraint violating interactions might occur between the process and implementation designers. If all implementation designers were under one authority, or if all processes were known to be cooperative, then the protection responsibility could be delegated to them. Since we cannot assume either of these situations to be true, the only common designer in all cases is the network designer. The network designer thus is the only agent capable of fulfilling this need. A need for network self-protection exists if the processes are to be well-defined.

Ensuring that the network design meets the constraints is an obvious requirement of the network designer. The process and implementation designers cannot be held responsible for this requirement since neither one of them has enough global information to know if the network meets the constraints or not.

Formally defining the network must be a responsibility of the network designers since they are responsible

for the design in the first place. A formal definition is required if the network designers are to expect it to be unambiguously understood by the implementers and the network users. It must also be formally defined if the designer is to know that it satisfies the design constraints.

> Design Constraint D3 - The implementation designers are responsible for the design, construction, and maintenance of an optimal equivalent system.

An equivalent system is one in which the processes are isomorphic to the processes as formally defined. Constraint D2 recognizes that the network will be implemented and that, according to constraint C2, some agent must be held responsible for the implementation. Each implementation designer is uniquely capable of determining what the optimal implementation at that facility will be. Only the implementation designer knows what physical resources are present or can be obtained at that facility. Each implementer must first design an equivalent system according to its own management constraints, then construct such an equivalent system, and finally maintain that equivalent system.

This constraint permits the local designers to continue to optimize their local implementations since they have complete responsibility for the equivalent implementa-

tion designer from moving an implementation to another physical machine or making modification to the current implementation. As long as the implementation remains equivalent, the local implementation designer is free to make any changes in the implementation deemed desirable.

The success that a network achieves in exploiting this delegation, of responsibility and authority to the implementation designer, will be highly dependent on the particular design chosen. It is particularly important that no one-to-one mappings be required between particular features of the logical network design and particular physical resources which might be available. Note that this says one-to-one mappings required by the design. One-to-one mappings which result by accident or the design of the implementers might be beneficial because of potentially increased efficiency, and is a choice of the implementers though and within their authority. Today's system designs often depend on such mappings and this makes them costly to move. If the "impressive improvements in technology" that Withington (Wi72) predicts are to be beneficial, new logical designs must provide for economical movement between facilities and between technology generations. Because of the responsibility delegation in this constraint, most of the interactions mentioned earlier which do not appear in the network as formally defined, will be the

responsibility of the implementation designers who will have authority to control them in any way that optimizes their own goals.

> Delegation Constraint D4 - The process designers are responsible for all virtual interprocess interactions.

Virtual interprocess interactions are those process interactions which would appear if the processes were being run as formally defined, and do not include those additional interactions which might appear in the equivalent implementations because the implementation designer chose particular physical resources for his implementation.

Constraint D4 follows from C2 and C4. The implementation designers cannot be held responsible for virtual interprocess interactions since each implementation designer may be under a different management and none of the constraints prohibit interactions between processes residing in different network systems. The network designers cannot be responsible for all virtual interprocess interactions because they cannot hope to know what all processes will be designed to do. The only designer remaining is the process designer. Even if it were not a constraint which is implicit in the postulated constraints, delegating to process designers the responsibility for all virtual interprocess interations would be desirable from a point of view of generality.

> Delegation Constraint D5 - If an implementation designer has an unsolvable problem due to boundedness which effects the processes in the network, the effect on the process state structure must be well-defined so the process designers can ameliorate the consequences.

If the first postulated constraint had not included bounded systems, constraint D5 could have been glossed over by adding a statement, in the design, that the implementation designers would always do the best they could. This would have required implementation dependent process transformations without a common definition of the effect of such "failures" on the formal definition of the network.

Implementation "failures" are solely the responsibility of each implementation designer under D3, as long as they do not effect the virtual processes. When the implementation designer declares that a process has hit a bound (an implementation "failure" has occurred which the implementation designer declared will affect that process state), the "failure" is no longer solely the implementation designer's problem. If the solution were left up to each implementation designer, each solution might be different and the processes might no longer be well-defined. Since constraint C4 permits a process to delegate to other processes authority to control certain interactions, the process which hits the bound may not be responsible for the problem. The implementation and network designers may

not have sufficient knowledge to decide what action to take to ameliorate the problem, only the process designers may have the information necessary to do it. The flexibility that a process designer has to do so, will be highly dependent on the features included as design choices from a point of view of generality. Constraint D5 is intended to force the network designer to include sufficient features in the design.

> Delegation Constraint D6 - Network designers are responsible for the initial definition of the network and implicitly for all evolutionary paths. Implementation designers are responsible for the implementation of the initial definition and for any potential evolutionary path. Process designers are responsible for selecting which evolutionary paths will be followed.

Responsibility must be factored between the three designers (C2) and network modification must be permitted (C5), therefore a delegation constraint must be developed assigning responsibility for network evolution. The network designer is the only one of the three designers which can properly be assigned responsibility for defining changes in the initial definition. D2 requires that network designers be responsible for ensuring the network is self-protecting, meets the constraints, and is formally defined. This constraint extends C2 to include responsibility for all potential network evolution, and requires

that the initial network definition also define all potential evolutionary paths. When the implementation designer's responsibilities in D3 are extended to include the implementation of modified definitions, it is imperative that such designer be able to determine if the implementation will be able to support such modifications before it claims it has implemented the network. In order for an implementation designer to be able to make such a determination, all potential paths of network evolution must be defined as part of the initial network.

It is possible to factor the remaining evolutionary responsibilities between the two remaining designers. An implementation designer must support the initial network and all potential evolutions if it accepts the network. This must be so since the implementation designer is the only designer which can be responsible for implementing such changes.

Process designers must be delegated the responsibility for determining which path to take. The network designers fulfill their responsibilities when they define the initial network and all potential evolutionary paths. Implementation designers fulfill their responsibilities when they implement the initial system, including modification operations for all potential evolutions as defined by the network designers. The process designer is free to

choose the particular path of evolution that it wants to take. Under C5, process designers are the only designers which can evaluate the potential effects of such modification on the processes. D4 delegates process designers as responsible for virtual process interactions. Since modification of the network may cause virtual process interactions (processes may behave differently after a modification than before), the process designers must choose the modification to be done. Neither the network nor the implementation designers have sufficient knowledge to decide which modifications to make. It is possible to delegate the authority to process designers to choose the path, since it is the network designer which designs the operators that process designers will cause to be executed. The network designer must ensure that any choice of modification operators will always be a permissible one.

Implementation Designer Constraints

This section is included primarily for completeness. Since it is not the intention of this thesis to look at implementation problems nor to constrain the implementation designers more than necessary, the constraints dealing with equivalent system implementation, modification, and boundedness are sufficient for our purposes. Each particular implementation designer will have to develop further imple-

mentation constraints when designing an equivalent system. Responsibility and authority for meeting the constraints and design decisions has been factored and future implementation designers will be responsible for this part of the design. The remainder of the chapter will look at further constraints on the process and network designers.

Process Designer Constraints

This section will look at some of the constraints which must be placed on process designers to ensure that their actions remain within the postulated constraints.

> Process Constraint P1 - Processes must be arranged in a tree structure defining their hierarchy of responsibility over descendent processes.

The network design must permit processes to delegate to other processes their authority to control process interactions. The delegating process always retains responsibility for that interaction and authority to control the process to which it was delegated (C4). There are many interactions over which process designers have authority (D4, D5). The possible process interactions mentioned earlier included such things as birth and death, allocation of logical resources, and logical expectations.

Implicit in constraint C4 is a hierarchy of processes which defines their relationship with respect to responsibility and authority for controlling other processes.

Each process in the hierarchy will be held responsible for its behavior to processes above it, even while delegating authority to processes below. There appears to be two alternative ways that such a hierarchy could exist. It could be as a general graph structure with the various threads being the different responsibilities and authorities, or the more restrictive form of a tree structure with the threads of responsibility defining the tree.

First, any particular chain of responsibility and authority cannot be permitted to double back on itself if the network is to guarantee well-defined processes. The concept of having authority over someone who has authority over you is incongruous. If it were permitted, the network would have to arbitrate conflicts, which it cannot do because it does not have sufficient knowledge of what the processes are trying to accomplish. The only alternative is to have a non-intersecting thread of responsibility.

This restricted graph would still permit there to be more than one process responsible for another process. If one process has more than one process responsible for its behavior, it would be impossible to ensure that there would never be any conflicts of interest between these authorities. When one of the responsible processes executed its authority, it might affect the process, for

which it is sharing responsibility, in a way which would affect the interactions over which the other process has authority. Such conflicts of authority would require arbitration which cannot be allowed if processes are to be well-defined.

The remaining alternative is to permit processes to be related only in a tree structure defined by their responsibility and authority over descendent processes. Since there is at most one immediate responsible agent, it can have complete and unique authority over the processes for which it is responsible. Although this is a constraint on process behavior, it produces significant freedom for the network designer because there is no need to be concerned over whether or not a process has a certain authority over another process. The designer must only be concerned with providing mechanisms which a parent process can use to exercise its authority over a child process, and not be concerned about the merits of the use of such an authority.

> Process Constraint P2 - A unique root process must exist representing overall process management.

Constraint D4 points out that it must be the process designers which control all virtual process interactions. The previous constraint points out that process

authority and responsibilities exist in the shape of a tree. An extension of the above arguments leads to constraint P2. If interactions between processes in different trees are to be permitted, then there must be an agent responsible for such interactions. Both trees must therefore be sub-trees of some common ancestor. It would be an arbitrary constraint on process behavior to say that two processes cannot interact if they exist in different trees. Any such root process may produce sub-trees which it chooses to isolate and consider independent trees if desired. This root process is thus unique for all of the processes in the network.

> Process Constraint P3 - No processor race conditions can be permitted to effect a process state transformation during that transformation.

This is a direct consequence of C1 which says that the design must have well-defined processes. If the design did permit processor race conditions to exist within a process state transformation, the design would no longer satisfy constraint C1. The actual mechanisms to be used to prevent such processor conditions from occurring are numerous, and the ones chosen for a particular network design may vary according to the influence of other design choices. This does not rule out the inevitable race conditions arising from the receipt of interprocess

messages by a system processor.

> Process Constraint P4 - Operators available to process designers affect only the containing process state.

If nonlocal process state transformations (transformations on process states other than the one that contained the operator that was executed) were permitted, constraint C1 could be violated, unless all such nonlocal transformation could be predicted, accounted for, and controlled by a responsible agent.

There are several reasons why these requirements would be extremely difficult to guarantee in a network design. If nonlocal process transformations were permitted, each implementation designer might be forced to make sure they could not cause processor race conditions for the effected process state. Even if such interactions were in general constrained to processes within the same system, the use of multiple physical processors might be difficult. It would also introduce an arbitrary constraint on process designers.

Such interprocess transformations, if permitted, would be a form of interprocess communication (in effect, shared space). Equivalent behavior can be achieved by the process (which would have initiated the interaction) transmitting a message to the other process. The message

then could cause the desired transformation as an operation local to the other process state. The final network design, if general enough, can provide these equivalent operations without the inherent overhead that would be required if nonlocal process state transformations were permitted.

Process Constraint P5 - Multiple processes may be transformed in parallel.

Constraint P6 is an obvious consequence of the first postulated constraint which says that the design is to consist of a network of digital computer systems. It would be inconsistent to expect the network as a whole to operate sequentially. At a minimum, processes in different systems must be permitted to execute in parallel. It is the responsibility of the network designers to define parallel transformations, and allow process designers to determine whether or not to make use of this facility. It would be improper at this time, though, to make constraint P5 any stronger.

There is nothing in the postulated constraints which would force the design to include parallelism in each system in the network. Although this might appear desirable, it must be included later as a design decision, not here as a design constraint.

Process Constraint P6 - Distribution of processes over the network must be per-

> mitted and each process state must be uniquely contained in, at most, one system at a time.

Constraint P6 is implicit in postulated constraint C1. It would make no sense to have a network of systems and then restrict all processes to exist in the same system. Process designers should be able to choose how to distribute processes in the network, and constraint P6 says that such a distribution is possible.

A process would not be well-defined if it were contained in more than one system at a time since each system would proceed independently in parallel. There is no assumption made about the relative speeds of the systems in the network which means that nothing could be said about the rapidity with which the disjoint process parts were proceeding. Since a process is a sequence of process states, the process becomes not well-defined when more than one system is transforming it because the sequence of process states would not be well-defined. Process designers are responsible for splitting a process into multiple processes so they can run on multiple systems.

> Process Constraint P7 - Interprocess communication must be permitted and be subject to parental control. Synchronization between processes is the responsibility of process designers. Processes residing in different systems may be synchronized only using interprocess communication.

If it were not for postulated constraint C4 (process delegation of authority) the network could be designed as a simple time sharing system where no interprocess interactions were permitted. Constraint C4 implies that there will have to be some interprocess interactions if the process designers are to fulfill their responsibilities. Since process transformations must be local to the process (P4), the only remaining means of meeting constraint C4 is to permit interprocess communication.

That such communication be subject to parental control is an obvious requirement, stemming from constraint C4 and developed in D4. Process designers are responsible for all virtual interprocess interactions and therefore must be responsible for interprocess communication. The need for a process tree hierarchy of responsibility (P1) puts the control of such interactions in the hands of the parent.

Process synchronization is an interprocess interaction for which process designers must be responsible under D4. Network and implementation designers cannot always know if two processes in the network are synchronized, either intentially or by accident. The definition of digital systems under constraint C1 says nothing about the relative speed at which process steps in two systems are carried out. Synchronization or lack of it between

systems is not a property of our network since it depends on the implementation. The particular allocation of hardware used by implementation designers for its optimal equivalent systems (D3) might have multiple logical systems implemented on one physical system, or might use multiple physical systems to implement one logical system. Without placing an arbitrary constraint, to implement either synchronous or asynchronous systems only (which we choose not to do), synchronization between processes in different systems can only be ensured by the processes themselves using interprocess communication.

> Process constraint P3 - An "accountable object" chain must exist, and processes must be able to restrict use of these accountable objects when sub-allocated.

Accountable objects are logical resources which processes may allocate to their children. The network design provides services for the accounting of these objects, including the return of them if a parent wishes them back and enforcing constraints on their use.

A process tree of responsible agents alone does not completely describe the implications of postulated constraint C4. It does not provide sufficient services for a process designer to use to control all virtual interprocess interactions. There are several interactions with respect to logical resources which the previous

constraints still permit to go uncontrolled. If a network is to permit processes to delegate authority, then this authority is a logical resource whose accounting and suballocation is something that a parent may need to control. The constraints make no assumptions about a child's cooperation with its parent. A special mechanism must therefore be provided which permits the parent to fulfill its responsibilities even if a child becomes uncooperative. The process tree defines responsibility, but does not constrain how deep the tree may be or how many children a particular process may have.

Although it is the responsibility of the process designers to decide what and how much authority to give to a child, the network design must provide remote enforcement services for accountable objects. At this point it is not important what is being passed, but only that certain accountable information must be passed. For example, communication must be subject to parental control (P7) and certain accountable information may be necessary to accomplish this. Since the children may not be cooperative, the network must provide use restrictions and accounting mechanisms, which may be invoked to protect these objects from the actions of a child.

These items (whatever they may logically represent) become accountable objects to the network. Since there

are no restrictions on how deep a tree may be or how many children a particular process may have, a constraint is required which provides sufficient facilities to process management to control the use of these objects. Given the existence of an accountable object, the network, if it is to perform its functions, must be able to locate the object. The network design must provide for an accounting chain (to the object from the process responsible for it) which is not subject to the desires of the child process. In addition, if a process designer is to delegate some of its authority, then mechanisms must be provided which allow restriction of the use of these accountable objects in addition to just accounting for them.

Network Designer Constraints

The previous section developed some of the constraints on the system design due to process designer responsibilities which can be derived from the five postulated constraints. This section will look at some of the constraints which must exist to permit the network designer to fulfill its responsibilities.

> Network Constraint N1 - The network designer is responsible for providing a set of invariant names which can be used by the process designers for communication and control.

Since different implementation designers, if not

restricted, might use local names for control and communication, process communication might become implementation dependent. If process designers created the names, they also might not be consistent in name creation. The only common designer capable of defining a set of invariant names, which can be used by the processes for communications and control, is the network designer.

> Network Constraint N2 - The network designers are responsible for changes in the network formal definition.

The fifth postulated constraint introduces the possibility of augmentation into the design, D2 designates the network designer as responsible for the network's definition, and D6 designates the network designer as responsible for all evolutionary paths. The network designer must also fulfill its responsibility by constraining evolutionary changes in the formal definition, before the implementers can actually augment the system.

> Network Constraint N3 - Sufficient services must be provided, by the network designers, for the process designers to accomplish their purpose, including resource accounting and intersystem communication.

The particular services provided by the network designers will be designed according to how they deem the network will be used. Resource accounting and intersystem communication are services already mentioned spe-

cifically, but there will be many more. This is a requirement since each designer must have sufficient authority to restrict the effects of the interactions for which it is responsible. If the process designers are to be responsible for all interprocess interactions, then the network must provide common services through which they may exercise this authority.

> Network Constraint N4 - Network modification operators are meta-operators and must first ensure the modification does not violate any of the postulated constraints and then either act as a no-op if it is one of the permissible paths of evolution or fault if it is not.

Network modification operator - A formally defined operator which the process designers can cause to be executed. It is intended to cause the network definition to be modified and then implemented by using locally defined procedures.

Meta-operator - An operator whose effect is not represented by a change in the process state. (In this case it changes the network definition.)

No-op - An operator which does not change the process state other than to advance the "program counter."

It is clear that such modification operators must be meta-operators since they operate on the network definition and not on the process state. That the operators must prove the modification does not violate any of the

postulated constraints follows from the delegation of responsibility constraint D6, where network management is responsible for all evolutionary paths. The only way that the network management can fulfill this responsibility is to design the operators so they ensure the modification does not violate the constraints.

The results of process management executing such an operator, must be either a no-op or a fault in the executing system. The execution of a modification operator may cause interactions which are the responsibility of each of the three designers. The definition must be changed, the new definition must be implemented, and finally the process must be told it succeeded or not.

If the operation had any affect on the process itself other than fault or no-op, then the implementation management would have to ensure that the process remained well-defined. All system steps would have to be designed and implemented so the execution of a modification operator during one of the steps would leave the executing process well-defined (this might be possible), but even worse, it would have to leave all other processes (also executing during that step) well-defined. This is clearly not the responsibility of either the process or implementation designers.

Since the network designer does not have complete

knowledge of what either of these two designers will always be doing, the only guaranteed non-race situation is to consider it a no-op in process transformations. The process executes the operator and either goes on with a no-op or faults (well-defined operations during that system step). The definition is either changed or not depending on the validity of the operation, and finally the implementation designer changes the implementation between system steps when all processes are not undergoing transformations and then starts the next system step using the new definition.

> Network Constraint N5 - The effects of network evolution, other than the addition of a new system to the network, must be local to the system within which the operator was executed.

This constraint follows directly from the fact that we are interested in digital systems (C1), the factoring of responsibility, and maintaining well-defined evolutionary processes. To say that the effects of evolution were other than local to the system in which the operator was executed would require synchronization between two systems, a violation of the concept of digital system. This would not rule out, of course, a process in one system executing an operator which caused a message to be transmitted to another process in another system. That

process could then execute the modification operator and cause that system to be modified since the final modification is local. The exact procedure again will depend on the particular modifying operators provided by the network designers. The addition of a new system cannot be considered local since there is no such system in the first place.

> Network Constraint N6 - The network design must permit process designers to limit the effects of a network modification operator to the process executing it. Process designers must be able to uniquely control the right to execute non-commuting modification operators (if any exist).

Constraint N6 follows directly from postulated constraints C1 and C5. Process designers must have responsibility and authority to control the effects of network definition modification on the process interactions for which the process designer has been designated responsibility. This is not quite strong enough to guarantee the well-defined processes as required by C1. Because the modification operator is really a meta-operator (N4), there could be interactions due to two processes simultaneously executing non-commuting modification operators which would result in non-well-defined processes or system.

C5 as stated could permit the effects to be limited

to the two processes executing the modification operator, but the resulting network definition might be different than either one of the processes expected. In order to guarantee that all processes will be well-defined, process designers must be able to limit the effects of the execution of a modification operator to the process executing that operator.

As a consequence of permitting the processes to suballocate their responsibilities the right to execute non-commuting modification operators (if they are permitted) must be controlled. If non-commuting operators exist, then two processes executing such modification operators could affect each other in a way that would make the operation have nonlocal effects and possibly produce processes which are not well-defined.

> Programming generality is the independence of an algorithm from the environment in which it operates (Denning, P.J. - De68, p. 6).

CHAPTER 3

NETWORK STRUCTURE DESIGN

Introduction

The primary goal of this thesis is the development and formal definition of a general machine independent network design which will support potentially uncooperative processes. In the previous chapter a set of constraints were developed which are intended to permit network management problems (relevant to the control of uncooperative processes) to be solved. Boundedness and network evolution were also included. Network designers are responsible for the design and formal definition of the network. Implementation designers are responsible for implementing locally optimal equivalent systems. Process designers are responsible for controlling all interprocess interactions between the processes in the virtual network, including those due to suballocation of the control of interprocess interactions.

This chapter will informally develop the major design choices relevant to process state structures and the transformations to be carried out on them. Within the bounds of

the constraints, many network designs could be produced, some of which would not provide what we feel is sufficient process generality. This chapter presents those design decisions, relating to such generality, which the networks defined by this design should provide. The design choices will be explained and where necessary shown to be consistent with the constraints of the previous chapter.

Since network modification must be permitted, a minimal system will be presented from which any particular network may evolve. The process designers determine which modification operators will be executed, and therefore retain significant control of the attributes of a particular network. A minimal initial definition also has the advantage of requiring the network design to include at least those network modification operators which are necessary to evolve interesting networks from the initial system. Although the sequence of presentation did not effect the validity of the constraints, the same cannot be said for the design decisions presented here. The development is intended to be complete enough to permit the reader to understand the essential characteristics of the network in all respects except its final representation. A formal definition of the network will be found in Appendix C.

There are several informal design principles which have been used in this chapter. An attempt has been made to provide facilities which process designers can use to con-

struct processes which implement their decisions concerning the control of their application processes. The network design should not build in particular strategies whenever it can be avoided, particularly if it limits the freedom of process designers. The tools provided should be useful and convenient so that interesting applications can be produced and properly managed. Each process in the network should be able to make finite progress, independent of what other processes in the network are doing.

In addition to providing a rich programming environment, the design should be general enough to be used as a model to compare current systems, and should be an easy framework for communicating ideas. To accomplish this the concepts should provide generality of expression rather than a list of options which tend to be confused. In order to maintain generality, the introduction of features reflecting specialization for specific applications or particular physical assets should be minimized. Where possible, placing constraints on the design of the programming languages should be deferred until they are required.

Although efficiency with respect to current systems is not a primary goal, process generality features should not be introduced without hope of reasonable implementation. Invariant network features should be identified so implementation designers can exploit them for efficient implementation. Intuitive notions of efficiency should be brought

into play when choosing between design decisions, particularly if they concern implementation on current systems.

Subprocessor and Expression-State

> Design decision A1 - Each system processor in the network will contain a set of identifiable subprocessors including at least the following three: GP (General Programmable), AO (Accountable Object), and PR (Process state Receive and transmit).

A subprocessor is defined as an operator of the system processor that is applied to a single process state. It may be designed to produce an arbitrarily complex process transformation, using only information within that process state. Process designers request the system processor to apply a particular subprocessor to a particular process state. The execution of a subprocessor only transforms the process state to which it is being applied and does not involve message transmission or receipt.

All operators of the system processor, including subprocessors, which are to be applied during a particular system step, are applied in parallel. The system step does not complete until all operators being applied during that step have completed. The system processor performs all operations not local to the process state (the transformation involves information in addition to that contained in the process state). For example, it delivers messages to a process state for disposition by a subprocessor, and dispatches messages after they are generated by a subprocessor.

The introduction of the concept of subprocessor into the network design provides a freedom to delegate responsibility for the design of the languages to be interpreted by the subprocessor to future subprocessor designers with a minimum of constraints. The network designer still retains responsibility for designing the network so process designers can control interprocess interactions. The subprocessor thus becomes the user programmable part of each system processor. The design of the language interpreted by a subprocessor will define the way processes may be formed by defining the transformations on process states.

Since processes are not assumed cooperative, it is convenient to limit the effects of the interpretation of such languages. The use of a subprocessor, whose operations can only be local to a process state, permits this. The system processor then performs all operations not local to a process state and can be designed to provide a well-defined interface for all operations which are not strictly local to a process state. The network designer need only constrain, not define, subprocessor operations which are always local to a process state. All non-local services of the system processor can be invoked implicitly by the subprocessors, and be invariant across the network.

An equivalent system processor could have been designed which did not contain subprocessors. It could be augmented with equivalent transformation operators and

proper sequencing and control functions to produce processes equivalent to those produced using subprocessors. Subprocessors simplify the network design. Without them, the system processor is cluttered with details which are not relevant to its basic operations of providing services not local to a process state. Understanding those operations which would have been logically grouped as a subprocessor is made more difficult because of the additional potential interactions they would have with the other operators of the system processor. Since a process state is observable once each system step, the observed process would be cluttered with the intermediate steps which would have been hidden by a subprocessor.

The designer of a subprocessor has certain freedoms that would not be present if all of the subprocessor functions had to be integrated into the system processor. Only the system processor, and the interactions of a subprocessor with it, need be understood by that subprocessor designer. Nothing need be known about the way other subprocessors operate during execution. It is assumed that all network designers will meet the constraints and design decisions. Without subprocessors, their responsibilities would include a much larger part of each system. All systems in the network would not necessarily provide the same set of subprocessors. The interactions to be prepared for, if the design did not use subprocessors, would therefore not necessarily

be the same in each system, complicating the problem even more. Many more constraints and design decisions would have to be introduced into the network to ensure that it supported only well-defined processes.

A subprocessor designer need not be concerned with having its operations interrupted by incoming messages, or the effect of the transmission of a message to another process whose subprocessor is in the middle of a process step. Nonlocal operations are done by the system processor. Buffering or other techniques can be used by a subprocessor designer during the execution of a process step since such information cannot be affected by anything outside the subprocessor until the step has been completed. The network designer thus guaranties the locality of process state transformations with minimal constraints on subprocessor designers.

Subprocessors are applied by the system processor to the process states, in the same manner as any other operators, and must be identifiable (recognizable and named) within the system processor. Process designers must have control over the sequencing of subprocessor applications, uniquely specifying the different subprocessors to be applied. The system processor, if the subprocessors are identifiable, can be designed to apply the proper subprocessor on command by process designers without caring what the many different processes in the network are intending

to do.

Each system in the network will have at least a GP, AO, and PR subprocessor which are common to all systems. The network designer cannot hope to know all the subprocessors present and future process designers will want to use. Therefore, although these three common subprocessors provide all initial network services, it would be arbitrary to define a set of subprocessors which would be available at each system in the network and then prohibit the introduction of any others. One of the purposes of introducing subprocessors into the network is to defer to future designers the development of subprocessors and the languages they interpret. The at least in design decision A1 is intended to permit the introduction of additional subprocessors. One of the network evolutionary steps that the design will permit is the introduction of new subprocessors into particular systems of the network.

The GP subprocessor is the general programmable subprocessor which is common to each system in the network. It provides all initial network services to process designers. The GP is programmable to ensure that process designers may invoke services to create useful processes, thus fulfilling network management's responsibilities to provide for programmable systems. The GP must provide all such services by interpreting a very general programming language since it may be the only programmable subprocessor in some systems.

Without at least one common programmable subprocessor available at each system, process transportability would not be very useful. The GP, since it is common, will provide process designers with one language whose interpretation is invariant in all systems in the network. If a particular system provides for additional programmable subprocessors, the process designer must make the choice of whether or not to use the GP. The GP is therefore not the "universal" language, but only a guaranteed "common" language.

The AO subprocessor is common to all systems in the network and has been delegated the job of fulfilling all network responsibilities with respect to accountable objects (N3 and P8). By delegating these responsibilities to a subprocessor instead of fulfilling them as part of each system processor design, the network designer is able to defer binding the structures used for accountable objects, while still having sufficient guarantees that the network responsibilities will be satisfied. The network design can proceed without getting involved in the semantic and syntactic problems involved with accountable objects and their use. The AO subprocessor will not be programmable in the general sense of the word, but will provide services to the programmable subprocessors of the network. It must be designed to carry out its responsibilities cooperatively with respect to accountable objects, even though particular processes may try to be uncooperative.

The final subprocessor which must be common to each system in the network is the PR (Process state Receive and transmit) subprocessor. The PR will carry out transformations on the PR process, whose function will be to receive both process states from other systems and requests for process creation and transmission from processes in its own system. If the request is to create a new process or if it is a process state from another system and the process state is valid to run in that system, the PR process requests the system processor to start the process. If the request is to move a process to another system, the PR process requests the system processor to transmit the process state to the PR process in the destination system.

Validation of requests for process creation and movement is simplified by requiring that within the network systems only the AO subprocessor be able to generate them. Since the AO subprocessor is designed to be cooperative, the PR can be designed to assume all requests, even from other systems within the network are valid, provided communication is designed so the source is known. Requests from foreign systems would present a verification problem since nothing may be known about such foreign systems. A PR process will therefore be designed to receive requests only from processes within its system or from other PR processes in the other systems of the network. The PR functions can easily be defined as a sub-

processor and delegated to future designers. An example PR subprocessor is defined in Appendix C.

> Design decision A2- A process state is defined by a set of identifiable, typed, possibly related expression-states, and associated information necessary for both subprocessor allocation and interprocess interactions.

A process state will have identifiable substates which will be called expression-states. Expression-states contain the state information associated with the execution of a subprocessor. Program interpretation information along with data structures and their associated values would be included. Expression-states are accessed and transformed by subprocessors when they are applied to the process state.

Without the requirement that expression-states be identifiable, subprocessors might not be able to distinguish one expression-state from another. Accidental subprocessor operations on expression-states other than the ones it should be accessing might occur, with possibly undefined results. In order to ensure that illegal process states did not occur, the design would have to place additional constraints which would restrict the generality available to process designers. We could have said that expression-states could not interact. There would then be no need for more than one expression-state in a process state at a time. This solution would not allow many conventional relationships such as that maintained by coroutines.

We could have restricted the process state from containing more than one type of expression-state. The whole process state structure could have then been made subprocessor type dependent. The state structure recognized by the subprocessor could then be made to define different expression-states. The system processor would only have to know which subprocessor to apply and the subprocessor designer would define all permissible operations on the process state. Although this is simpler, it is not as general as permitting multiple kinds of related expression-states to exist within one process state.

Once the design permits a process state to contain multiple kinds of expression-states, all subprocessors can no longer be guaranteed to recognize all of them. For example, a process state might contain three kinds of expression-states related in a tree structure (root and two branches). The appropriate subprocessor for either branch might be able to interpret the root expression-state, but not the expression-state at the other branch.

If the design is to support well-defined processes, each subprocessor must recognize when it has referenced a new expression-state. This prevents a subprocessor from accidentally transforming an expression-state because an expression-state boundary was unwittingly crossed. The network design must therefore provide expression-state delimiters in a process state so that it is possible to recog-

of the network design.

Dealing with logically asynchronous systems has the advantage that individual system designers and implementers need not be concerned with what other systems are doing, except for the asynchronous communication. On the other hand, if the users want logical synchronization between processes running on different systems in the network, they must accomplish it using asynchronous interprocesses communications, rather than by relying on synchronization of the steps of the systems (intersystem lockstep is not defined in the network).

The operations of each network system are carried out by a system processor which includes a set of subprocesors that carry out transformations on the expression-states within a process state. System processor execution takes place in three phases: subprocessor allocation, subprocessor application, and message dispatching. Each process in the network has a single process state (contained in one system at a time) and is said to undergo one process step each time a subprocessor is applied to that state. A process step becomes the only invariant measure of "time" in the network.

The system processor will use information contained in the process state to apply subprocessors, but will apply only one subprocessor to a process state each process step. Since only one system processor has access to a process state at a time, subprocessor application becomes an opera-

tion local to that system.

Subprocessor designers do not have to be concerned with race conditions due to simultaneous access to a process state because only one subprocessor is applied to it each process step. The design of subprocessor operations which require more than one process step, must still solve race problems. Interrupts might occur between process steps, causing a different subprocessor to be applied, or the same subprocessor to be applied to a different part of the process state. If process designers choose to permit interrupts, they must take the appropriate actions to ensure that the occurrence of an interrupt does not have an improper effect on the process.

Each process state is composed of a set of typed expression-states (ES), some of which may be paired with control-points (CP) to form ES/CP pairs. All transformations local to a process state are carried out by the subprocessors. An operation is local to a process state if the process state contains all of the information necessary for the operation and if the effects of the transformation are contained in that same process state. Expression-states contain all of the information necessary for subprocessor operation. All operations which are not local to a process state are carried out by the system processor using information associated with the control-points.

The system processor only accesses control-point

information and not information in expression states, therefore the internal structure of expression-states can be made subprocessor dependent. Certain expression-states may be interpretable by only a subset of the subprocessors in the network. Subprocessors are constrained from arbitrarily accessing a type of expression-state other than their own. If a subprocessor designer wishes for it to have access to another expression-state type, then the subprocessor must be designed to abide by the conventions laid down by the designer of the other expression-state.

The second part of an ES/CP pair is a control-point. A control-point has associated with it the information necessary for subprocessor application (type of subprocessor to be applied, priority, demand bit) and interprocess communication (unique name, destination buffers with arm bits, enable bit). Control-points provide a common interface for interaction between subprocessors and the system processor.

Process designers use the demand bit to request that a subprocessor be applied to that ES/CP pair. It may be set locally by a subprocessor operating on the containing process state or because of the receipt of a message destined for that control-point. If more than one ES/CP pair has its demand bit on, the system processor uses the priority associated with each control-point (priority assignment is static and arranged so no intraprocess conflicts can occur)

to determine which pair should cause a subprocessor to be applied, and then uses that pair's control-point to determine which subprocessor to apply.

Control-points are recognizable by all system processors so that it is possible to move a process from one system to another without concern over how to cause subprocessors to be applied. (Note: All systems are not required to contain all subprocessors, each system processor applies only those it knows about.) The user is thus free to move a process from one system to another to gain access to special subprocessors.

Interrupts and faults are clearly distinguishable in the design. An interrupt occurs within a process because the demand bit of a higher priority control-point was turned on. Such occurrences only affect the decision of which subprocessor to apply on the next process step and do not cause pre-emption of a subprocessor executing a process step. Faults, on the other hand, occur during a process step and cause an executing subprocessor to change its program interpretation sequence. Faults have no direct effect on which subprocessor will be applied on the next process step.

Single interprocess messages are generated by subprocessors during the subprocessor application phase of the system processor. They are placed in the interface buffer along with a destination address (CP name, buffer name).

During the message dispatching phase, the system processor will remove the message and transmit it. The destination system (containing the process state with the destination CP in it) will look at the appropriate buffer (a static number are associated with each control-point and are named individually) to see if it is armed. If the buffer is armed, the message will be buffered (overwriting any previous message) and if not armed the message will be disregarded.

Also associated with each control-point is an enable bit which enables sending messages to cause an interrupt. If the control-point is enabled and of appropriate priority, the system processor will place a pending message in the interface buffer on the message delivery phase and apply the appropriate subprocessor on the subprocessor application phase.

Since control-point names are unique and serve as destination addresses for messages, it is not necessary for the sender to know either the system in which the destination process resides or even the process itself. Processes can thus be moved between systems without a need to arrange a new communication link. The control-point itself could even be moved to a different process without a transmitting process knowing the difference.

The relationship between the processes in the network looks like a tree and defines their hierarchy of responsibility and control over descendent processes which

may be distributed arbitrarily over the network. Each network system will have an AO subprocessor (identical in all systems) which will account for certain objects, and enforce restrictions placed on them; each process will contain an ES/CP pair upon which the AO subprocessor operates. The AO control-point always has the highest priority in the process which makes it impossible for the AO services in a process to be uncooperative with its parent. The AO subprocessor cannot be locked out since a message to the AO ES/CP pair will always interrupt all other pairs in that process.

All accountable objects will be maintained in the AO expression state. This gives the AO subprocessor designer control over the mechanisms used to access them. Some of the accountable objects may represent rights to carry out certain operations. Each control-point is an accountable object. This permits a parent to control which subprocessors a child can use since subprocessor type is associated with the control-point. A parent can control a child's message transmission capability by rights to transmit to particular control-point buffers. Rights to move a process or execute a particular subprocessor operator may also be included. In addition to rights, accountable objects may contain logical objects such as files. The set of accountable objects can be thought of as representing the logical capabilities of a process.

In addition to the AO subprocessor, there will be at least two other subprocessors (GP, PR) contained in each system. The GP will provide general programming services and access to all the initial network services by interpreting a common language. Without one such common subprocessor in the network, process intersystem movement might not be very useful.

The PR will provide process state receipt, transmission, and creation services in each system. Communication between the processes in the network and systems foreign to the network is also permitted using normal interprocess communication mechanisms. The set of accountable objects may contain rights to transmit to control-points which exist only in foreign systems, and certain control-points will be permitted to have buffers to which only foreign systems transmit. No special mechanisms are required to permit a process to interact freely with some outside agent.

Use as a Model

The network was designed to be general in the sense that it contains the concepts of most other systems. This section will model three current systems using the designed network to show that such models exist and to relate the three system structures. These systems do not completely meet our needs (they were designed for different goals). TENEX (BB71) and the RC4000 (Ha 70, Ha 71a) system will be described first, as examples of systems which give users the

ability to create and control a "process" hierarchy. The ARPA network (CC 70, CH 72, TH 72) will be used as an example of a distributive network composed of different types of computers.

TENEX is a time-sharing system implemented on an augmented DEC PDP-10. It is the logical structure as seen by the user that is to be modeled, not the implementation of it. Three examples of the many alternative ways that TENEX could be described in our terms are shown in Figures 1, 2 and 3. Boxes in the Figures represent our asynchronous systems, circles represent our process state, and the lines joining them represent the process hierarchy. In Fig. 3 each set of parentheses represents an expression-state. We might first describe TENEX as a process tree distributed over a set of asynchronous systems, each system (box) containing one process (circle). (Fig. 1). Since we would thus have only one process in each system and since TENEX processes have no well-defined structure below process state (like our expression state), we would need no structure below the system process and would consider the user process to be the system process as well. The Monitor (running on the real system) can be thought of as the root process of a single process tree. The monitor process creates an asynchronous command system with an EXEC process in it for each new job, which then serves as the top of that job's process tree. We would not consider this system a TENEX

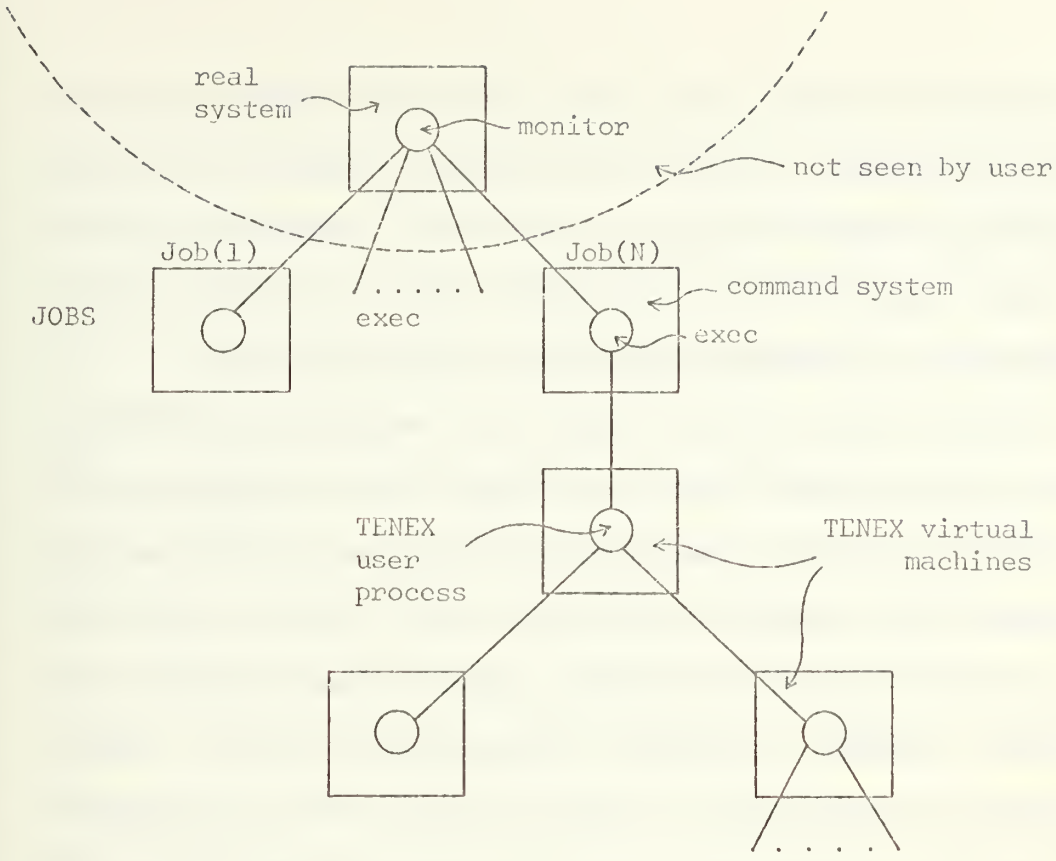


Fig. 1--TENEX Model 1

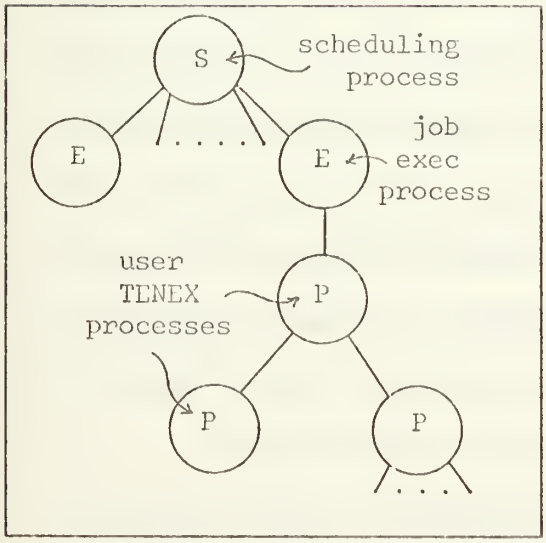


Fig. 2--TENEX Model 2

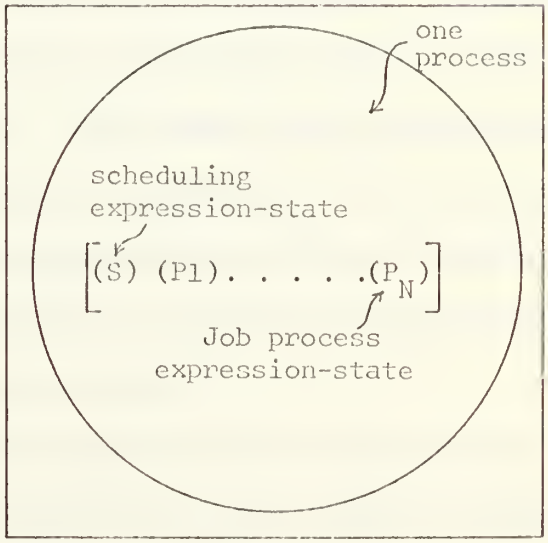


Fig. 3--TENEX Model 3

virtual system since from the user's point of view it interprets the command language and not the language of the TENEX virtual machine. For each job process below the EXEC we would create a system containing a TENEX subprocessor.

Model one puts each TENEX process in a separate asynchronous system. In model two the whole process tree is contained in one system with each TENEX process being represented by one of our processes, while in model three the whole process tree has been collapsed into one process, each TENEX process being represented as one expression state. Three subprocessors (schedule, command, TENEX) are contained in the single system in models two and three, instead of one subprocessor for each system as in model one.

It might appear that the multisystem model of TENEX is better than the other two since it maps each TENEX virtual system onto one of our asynchronous systems. Although this may be the appearance they wished the system to have, in fact it is not what exists. This model completely hides the presence of the time-sharing algorithm (we say nothing about the relative speed of asynchronous systems) and the sharing of memory by TENEX virtual systems (our asynchronous systems do not logically share memory).

In the first and second models we can explicitly represent the time-sharing algorithm by the introduction of an extra ES/CP pair (highest priority) in each process and a time-sharing process at the top of the process tree to

receive clock signals and monitor messages for controlling the schedule. The T/S process would interrupt the executing process by sending a message to the extra pair. The message could include the next process to run so that as soon as the demand bits and enable bits for any control-points in the process had been turned off, the extra pair would send a message to the extra pair in the next process to run.

One way of representing shared space in models one and two is to represent all space shared by TENEX processes as an object message and transmit it along with the wakeup message.This is a good model in the sense that it explicitly ties shared memory to access to a subprocessor, but it hides the relationships between the processes doing the sharing. Model three not only can represent the time sharing algorithm but also the sharing of space. The T/S pair would receive the monitor and clock information and manipulate the control-points associated with the TENEX processes to execute the scheduling. Since expression states within our processes are permitted to share address space and the system processor applies only one subprocessor at a time to a process state, model three provides an exact representation of TENEX in our terms.

In each of the three models the TENEX "pseudo" interrupt, as they call it, could be represented either as an interrupt or as a fault. If we used our interrupt mechanism, we would say that each TENEX process has two control-points

associated with it. A higher priority control-point would interrupt the process when pseudo-interrupt messages were received from the monitor. The lower priority control-point would be used for the normal processing, including the receipt of wakeup messages after certain monitor calls.

The alternative is to describe each TENEX process with only one control-point, representing the pseudo-interrupts as faults delivered by the implementation to the subprocessor being applied. The interrupt models nicely the fact that certain of the events are truly asynchronous with respect to a processes execution (from terminals, other processes) but the fault models nicely those events which are a result of a local execution (arithmetic). Since the TENEX virtual machine does not clearly distinguish between interrupts and faults as we have in our system, either solution would be sufficient as a model of TENEX. In all three models using the interrupt mechanism, a hierarchial subprocessor application scheme would be used to determine how to apply the subprocessors to the process states.

There are almost no accountable objects manipulated by job processes in TENEX since the time-sharing implementation attempts to allocate to each TENEX process its "fair share" of physical resources. A major difference between TENEX and our system is that we have introduced accountable objects and the AO subprocessor to provide a mechanism for a parent process to use if it wishes to allocate some of its

accountable object to its children while maintaining control over the use of them. The ability to receive pseudo-interrupts in TENEX can be thought of as an accountable object in the sense that a parent has the right to field interrupts for its children. One of the major consequences of not having accountable objects is that TENEX is not really able to support a hierarchy of operating systems.

There are several features of our network which are not present in TENEX. The TENEX virtual machine is as complex as their concept of system can get and in it there is no concept of lockstep process transformations which we get when multiple processes reside in one of our systems. TENEX also does not have the complex process states we can construct using multiple expression states (Model 3). The authors of "TENEX, a paged time sharing system for the PDP-10" (BB 71) use, as an example to illustrate the usefulness of TENEX's structures, a program wishing to wait for terminal input but only for a specified length of time. Two processes are created, one waiting for the input and one waiting for the timer. We could solve this same problem using two ES/CP pairs within the same process rather than two processes. The process could even be doing some additional processing using a third lower priority ES/CP pair and still be interrupted when either of the two events occurred.

Whereas we can model their system with its time-

sharing algorithm, they could not hide the fact that they are a time sharing system if they were to try and model our system. The TENEX structures are designed for single system interprocess interactions. Even these are highly constrained between jobs in the same system, whereas our normal interprocess communication structures are the same no matter what the residence of the processes carrying out the interaction. Since our processes are free to move between systems in the network, we have not permitted processes to share address space except through explicit transfer of the information (by the process not the system). With respect to our goals, TENEX is most lacking in the facilities provided for user management of potentially uncooperative processes.

The RC4000 multiprogramming system developed by Regencentralen was designed to "be extended with a hierarchy of operating systems. . ." (Ha 70). Resources are allocated recursively by a parent process to its children. The RC4000 system can be represented in our design as a single system containing one process state, similar to the third TENEX model (Fig. 3). Instead of the three subprocessors in the TENEX model we can describe the RC4000 system with only one. The single process state will be composed of a scheduling expression-state and one expression-state for each RC4000 process currently residing in core.

Since expression-states may share address space, the

decision to model the RC4000 system with one process state permits us to model explicitly the potential overlap of addressing capabilities between RC4000 processes in core at any given time (keyed access is used). Those RC4000 processes which are currently stopped would exist as data objects in the expression-state representing their parent process. This is the way the RC4000 system permits a parent to deal with them.

Our model would associate a control-point with each expression-state (RC4000 process), plus one attached to the scheduling expression-state which would have the highest priority. The association of a control-point with the expression-states representing RC4000 processes is necessary since each process may have messages directed to it by other processes in the system. Each RC4000 process needs only one control-point associated with it since there is no concept of our interrupt in their system, as the user sees it. Their internal interrupts are represented by our faults. "Wait" operations control RC4000 process execution by turning off the demand bit associated with the single control-point. After completion of the operation, the scheduler turns on the demand bit and execution continues where it had left off before the wait. It is this simple behavior of the RC4000 process that permits us to represent it as a single expression-state with one control-point attached.

If we knew that the address spaces of all RC4000

process were disjoint, then we would represent each one of them by one of our processes containing one AO expression-state and one processing expression-state. The AO expression-state would perform the parent's breakpoint operations, insure that resources were allocated properly, and perform the scheduling function as discussed in the TENEX model.

There are two additional ways we might represent shared space in our models and still use multiple process states, without sending a shared space object along with the scheduling message as was done in the TENEX model. We could consider the shared space to be maintained by either a "shared space" process or foreign system, and have the sharing processes communicate their requests to these agents. Although this would adequately simulate the effects of shared space, it would introduce the artifice of using interprocess interactions to accomplish it.

The implementation of memory sharing mechanisms usually involves either a foreign system (shared memory modules with conflict resolution on requests) or the process mechanism (monitor calls which insure that only one process at a time accesses the information). These models, of course, are simply models of low level system structures for normal hardware systems. A machine independent virtual system structure must not be sensitive to such low level details.

"P & V" operations can be modeled nicely using

either of these mechanisms, although in our system we could provide more control over the sharing since the process address spaces would be disjoint and we could prevent invalid entries. In our model, sufficient control over this potentially uncooperative behavior could be achieved by explicitly defining the shared space process to supply the information properly even if the P & V operations were not used. Uncooperative processes cannot be prevented from accessing shared space by using only cooperative "P & V" operations. This mechanism is thus useful but not sufficient for process control.

There are two important differences between the RC4000 system and TENEX with respect to their treatments of resources and interprocess messages. The RC4000 system makes explicit use of recursive resource allocation. Part of the procedure for creating a child process involves the explicit allocation of some of the parent's resources (always a subset). A TENEX process can control the execution and existence of its children, and field pseudo-interrupts for them, but it does not specifically allocate resources to them. All TENEX resource allocation is based on a "fair share" algorithm built into the implementation.

In our single process model of the RC4000, the resource allocation would be represented by creating the child expression-state with access to objects representing the proper allocation of resources. In our multiprocess

model, each process (representing a RC4000 process) would maintain the information about that process's allocation of resources in the AO expression-state, which we use for our recursive allocation of accountable objects.

In both the RC4000 system and our system, a parent process recursively allocates subsets of what it has, but there are several important differences between the mechanisms used. Our accountable objects are explicitly maintained in a reliable expression-state in each process and we provide a common subprocessor in each system to operate on that expression-state. By including the accountable object information as part of the process state, it automatically moves with the process. A transmitting system does not have to send an additional message to the receiving system concerning all of the accountable objects and information necessary for their accounting and protection. The common subprocessor reduces the complexity of each system processor and assures that each system will invariantly handle accountable objects. In contrast, the RC4000 monitor maintains the resource information and performs the operations on this information. The RC4000 system does not need to keep the resource information local to the process state since their processes do not move between systems. A special subprocessor is not really necessary since the resource function can easily be incorporated into the monitor without adding unnecessary complexity. RC4000 resources are not

really pre-emptible, other than by killing the child, and a parent cannot explicitly attach access restrictions to their use. A RC4000 process is created with a given set of resources and uses them at its discretion.

Interprocess communication is handled differently in the RC4000 system and TENEX. TENEX uses their pseudo-interrupt and a shared address space, but restricts such interactions to the processes of one job. An advantage of the TENEX mechanism is that the pseudo-interrupt permits a process to be told a message is pending, whereas the RC4000 system requires a process to ask for the message and if it is not there, go to sleep. The RC4000 system uses message buffers (a resource not part of a shared address space) and permits a process to communicate with any process it knows about. The advantage of the RC4000 mechanism over TENEX is that it does not require shared space; more than one message can be sent at a time; and no restriction is placed on how many different processes can be transmitting to one receiving process.

Our communication mechanisms have an interrupt ability and do not use shared space. We associate multiple buffers with a destination control-point and consider the right to transmit to a particular one of these buffers as the accountable object, not the buffer itself. We thus control the right to transmit to a particular destination while they only control the means of communication and not who can be

talked to.

Although both TENEX and the RC4000 systems are monoprocessor systems, the system structure they use could be extended to a multi-processor level. Appropriate subprocessor application rules would have to be instituted and all primitive operators implemented so they would leave the user processes well-defined, regardless of the number of subprocessors operating concurrently in the system. Most of the difficulties to be faced would arise because the user processes can share space and because of system race conditions for process state information. These are basic problems that drastically effect the structure and efficiency of such an extension. The extension to a network or system would be very difficult and would severely constrain user management of processes. In our system, process states are disjoint, only one subprocessor is applied to a process state each process step, and external interactions with a process do not occur during a process step. Subprocessor designers therefore do not have to be concerned with resolving race conditions for shared access since it never occurs.

We want to briefly look at computer networks. The most degenerate form of our network structures would be achieved by connecting a set of systems together without defining their system structures or placing restrictions on their interactions, other than with which other systems they can communicate. Such a network would be defined as

a set of autonomous interacting foreign systems. The network designers can say very little about the processes running in such a network. Only the set of systems with which a process can communicate would be known. Nevertheless, this degenerate network is a good model for the ARPA Computer Network (ARPANET).

ARPANET provides "for the interconnection, via common-carrier circuits, of dissimilar computers at widely separated" centers (OH 72). Protocols (agreements on the format and relative sequences of messages to be exchanged) were designed as a layered hierarchy so each level does not have to be concerned with the intricacies of the lower levels (CH 72). The lowest level of the hierarchy is the message exchange between the Interface Message Processors (IMPs), and the highest level is the user process to process communication.

In contrast with the degenerate network above, we can model the ARPANET IMPs as a network whose processes interact with a set of foreign systems (HOSTs). The IMP processes are fully defined by the network designers, the users have nothing to say about them. Although it would be possible for us to model the whole ARPANET at any one of the above levels, it is only the higher level we wish to model here. In our terms, the ARPANET HOSTs can be modeled as a set of foreign systems supporting interacting user processes. In order to understand the behavior of any pair of these

processes, we must model each foreign system processor down to the application of subprocessors. Those HOSTs operating under TENEX could be modeled using one of the models above. Similar models could be developed for each HOST. Since the process structure of HOSTs are not constrained, existing ones can be changed and new HOSTs with new process structures can be added to the network. Process behavior can thus be defined only locally to each system according to current structures.

In the ARPANET, process behavior is dependent on system residence. There is no standard concept of process step or interrupt since there is no standard system processor. The lack of standard structure requires independent study of each system model if there is a need to understand their behavior. Although this is simplifying from the implementers' point of view, it is not particularly useful from the users' point of view. In order for a user to take advantage of the asynchronous systems to construct an asynchronous multiprocess computation, the structures of each system to be used must be clearly understood, and local rules and conventions must be used to design and manage each process.

The ARPANET assumes no responsibility towards interacting processes other than providing the communication link to each operating system, just as our design assumes no responsibilities towards a process in a foreign system. There

are no ARPANET structures which permit a user to institute remote constraints on process behavior, and no notion of the RC4000 resource or our accountable objects. Processes in separate systems may communicate, but delegation of responsibility and authority between them must be purely cooperative and subject to local HOST constraints.

The ARPANET provides five interprocess communication primitives in each HOST system interface. These permit, via HOST operating systems, interactions between processes residing in different HOSTs. Since there is no standard concept of either system or process, a user must understand each local definition before constructing a multisystem computation. The effect of using these primitives becomes highly system sensitive.

Each ARPANET HOST has a set of send and receive sockets whose names have three parts: user number; HOST number; and AEN (a seven bit number plus one bit for send or receive). Socket names are thus unique throughout the network while providing a pool of sockets for each user at each HOST. In order to use these sockets, a process must interface with its local Network Control Program (NCP), which acts on its behalf to establish, break, and switch connections. Once a particular connection between two sockets has been made, the process is free to communicate using that connection until it is broken.

The ARPANET designers used the fairly static connec-

tion mechanism because they felt that processes would conduct prolonged conversations and not one shot requests. Although the NCP ensures that the process requesting use of a particular socket to make a connection has the same user number as the socket, once that connection has been made it is possible to move the connection to a process running under a different user number. This, at least, means that no other user may uncooperatively exhaust another user's pool of sockets, although the processes under one user number must still be cooperative in their use of sockets to make connections. There is also no mechanism, other than cooperation, by which a user may control the use of a connection once it has been made.

Thomas and Henderson (TH 72) describe a major disadvantage of the ARPANET connection mechanism as requiring a process to know the socket number that the other process will be using for its end of the connection before the connection can be made. This implies knowledge of which HOST the process will be in and the user number under which it will be running as well as the particular AEN to be used. In our design a control-points' name serves as the destination address, but although the name is unique, it is not bound to a particular system or user. Both the control-points themselves, and rights to transmit to particular buffers associated with control-points are accountable objects and therefore recursively managed by the process

hierarchy. Our design also leaves the actual making of a connection up to the implementers. If a process has the right to transmit to a particular control-point buffer, it can send a message without becoming involved with making the connection.

Since all system processors in our network use a standard concept of process step, interrupt, message delivery, and standard operations on control-point, the user is provided with a consistent concept of process and process step and a standard way to use the services of the network. An ARPANET user must be prepared to create all processes local to the system in which they are to run since different HOSTs may have different process structures and intersystem process migration may not be possible. Our design, on the other hand, provides explicitly for intersystem process movement and provides for at least one common subprocessor (GP) which will interpret a general language and provide all initial network services. This, of course, does not ensure that a process, using subprocessors other than those common to all systems, will run in any system to which it is moved. A user thus has an alternative to creating a different process in each system and exchanging control and data information between processes. A user can create a "multi-lingual" process, use the GP subprocessor for common services, and move the process from system to system to access special subprocessors.

The discussion here was intended to show how the user-level protocols compare with TENEX, RC4000 and our system. It should be noted that we could conveniently use these ARPANET protocols to implement our system structures at each HOST. Our system would thus become one more level of protocol in the ARPANET hierarchy. ARPANET deficiencies largely arise from the attempt to use such an implementation structure in place of the virtual network that is really required.

Further Work

It would be conceivable to answer questions concerning the generality of the postulated constraints by extending this work in several directions. One of the first extensions which comes to mind is to include not only systems as components of our network, but also other networks, giving us structured nodes and a hierarchy of networks. Questions concerning the relationships between processes in the various levels of this hierarchy could be investigated. The systems in one network level might be considered to be foreign systems to the processes in all other levels, or perhaps a logical interface process should be provided, through which connections between networks are maintained. Responsibility and authority problems begin to surface again, in particular the rights of processes in one network with respect to such areas as modifications and manipulations of accountable objects (Global vs. local rights).

The sufficiency of our constraints and the correctness of our design decisions when real time and continuous (analogue) processes are added to the system is also in need of investigation. Does our design remain invariant when used for these processes, and what can be said about process behavior? In the case of real time processes, the process state structures probably are not effected but the relative speeds of systems will effect real time process performance. In the case of continuous processes, any investigation must first define what is meant by continuous processes, how to represent them, and how to combine continuous and digital processes into one model.

The design in this thesis must be extended to include the subprocessors and their modification operators, before an operational network can be produced. The GP subprocessor is to be a programmable subprocessor common to all systems and must provide all initial network services within a framework of a general language. The design of the AO subprocessor must include investigations into classifications of accountable objects, operators for their accounting and maintenance, and the effects of process birth, death, and intersystem movement on them. Investigations into network modification operators must be continued to determine the forms they may take and the freedoms which can be permitted for modifications, including modifications which are not just augmentations.

There are several implementation questions which can be pursued based on this design. We can exploit our knowledge of the system to study the optimization of an implementation. Once a prototype implementation exists we can monitor the process behavior at both subprocessor primitive level and the system processor operator level to improve our implementation. We can then study independent of any particular implementation, the question of practical machine design including buffering and efficiency along with subprocessor augmentation, program type objects, and special subprocessors.

In addition to studying implementation questions we can abstractly study the properties of our formal definition. A parallel effort is being pursued (Jo 73) which is intended to derive and prove assertions that certain features of the state languages of systems (using the definition system in FH 71) remain invariant to transformation under that system processor. We will then be able to use these mechanisms to investigate the formal definition of our network and the processes it supports. We would hope, by studying the network definition itself, to verify and even formally certify our design from the point of view that certain of the designed system process state structures are proven to be invariant under the transformations carried out by the system processors. Our design has been "debugged" by test cases, using an interpreter for the formal definition

system, but this testing alone cannot be sufficient to certify that these invariances remain so under all transformation.

Once the network exists, it is clear that not all users will be permitted or even will want to use the full flexibility of our structures. Investigations will be needed to determine what user operating environments should be created by using appropriate processes in the process tree and distributed over the network systems. Using the hierarchy we can investigate distributed vs. central control, and various levels of user freedoms. We can thus study concepts of logical operating system structures and commonly desired services. Our network provides an operating system implementation language for such studies.

One final area of investigation is the development of application tools including the various application languages which should be provided in a network. We would like to develop a translator-compiler (written in the GP language) to take some of these languages and generate translators for them, whose output is the language interpreted by GP. This would permit such processes to be system independent since each system has a GP. We would also like to develop certain special purpose applications such as distributed data base and management information system, and user process heirarchies which interact with special purpose foreign systems.

Several of the above items are currently being investigated. The initial design of the GP subprocessor has

been completed (Fi 72) and an implementation of it exists written in a common subset of FORTRAN IV. A Ph.D. thesis (Co 73) is under preparation to complete the design of the AO subprocessor and another student is investigating modification operators (Pa 73). A single physical system implementation of one logical system is in progress which will then be duplicated within that physical system to give us multiple logical systems. A movement to a multiple physical system implementation of the multiple logical system will be attempted dependent on the completion of a small physical network being constructed at the University of Wisconsin. A research seminar (Spring 73) is investigating the questions of translator-compilers and defining one in the GP language to make it available in each system. Most of the other projects have been discussed and considered as future work but are not currently scheduled since they are dependent on the work currently underway.

Summary and Conclusions

The emergence of networks of digital computer systems has resulted in severe problems concerning their managability, generality and portability. A managable network design must provide for the resolution of conflicts of responsibility and authority. Generality must exist if the network is to be useful to many classes of potential users. Portability must be possible if the network design and user processes are not to be implementation dependent.

A sufficient set of postulated constraints on the network design were developed to permit a solution to these problems. Difficulties in network manageability result primarily because of the conflicting goals of the network, implementation, and process designers. The solution requires a clear factorization, between these three agents, of the responsibility and authority for controlling the behavior of the processes running on the network.

A design was then carried out within the postulated constraints to show both that a solution to the network problems does exist within these constraints, and that it leads to a design which it is feasible to implement and which is general enough to contain at least most of the current system structures. A set of derived constraints were developed to explore in more detail the consequences of the postulated constraints, and then a set of design decisions were developed which defined the generality of the structures to be provided the users. Finally several current systems were modeled using the network structures developed and the structures themselves were formally defined.

There are many conclusions which could be drawn from the work presented here. One of the most significant of these is that it was possible to accomplish such a general goal, with the result being a clear solution which seems to be implementable. This success appears to be a result of having undertaken the whole problem of manageability,

generality, and portability, and not approaching only one part of it. The particular factorizations of the problem made in this thesis were based on the constraints and design goals and not as a result of arbitrary or conflicting decisions as defined in current systems. Factorization of responsibility and authority was the tool which makes the solution possible. It does not appear that it is sufficient to approach these problems independently (for example, trying to solve the problem of portability while leaving current system structures unchanged and independent). It may be that because of the interactions of these problems, a global solution such as presented in this thesis, is the only possible path to a general solution. We have shown that it is a sufficient path.

As a result of having a set of machine and technology independent definitions which are at least capable of modeling current systems, it is possible to discuss system and process structures and compare them using the network design presented here. The vocabulary used is independent of the system being described, and presents a well-defined framework within which to define problems. The concept of a general process step provides maximum freedom to the user and designer, while providing sufficient rules to permit work to be actually accomplished. Processes defined to run on our network systems will provide portability not only between currently implemented systems but also between

generations of implementations of the current systems. This means that it is now cost effective to invest heavily in applications and application tools since they can survive changes in technology. Process designers can define local logical operating systems, including the management of accountable objects, in distributed computations which are not system dependent.

Many of the features of the design are interesting when taken by themselves and need not be applied only to a network. Control-points combine the functions of both communication and control (interrupt) and permit interprocess interactions to be directed to particular parts of a process state and not just the process itself. Subprocesses and expression-states in conjunction with control-points, permit the concept of process step to be generalized. The subprocessor designer is able to define its state structures and operations on that state, thus deferring binding of potential process behavior indefinitely since the introduction of new subprocessors does not affect current process behavior. The concept of accountable objects, and their allocation and control by the process hierarchy provides a clear mechanism for creating user operating systems. Evolutionary processes permit the deferral of the choice of a particular network, while providing a clear mechanism for generating it and for making adaptive changes to an existing definition. The fact that there is never any asynchronous

fighting over a process state by systems or subprocessors means that the implementers have no race conflicts to resolve. The process must, and can, resolve its own problems locally to that process. Finally, although the network was designed to provide mechanisms for the control of potentially uncooperative processes, process cooperation can still be exploited. Within a process state all operations except operations on accountable objects, are considered cooperative and limited only by the operators available in the subprocessors being used. If the processes wish to use normal control-point communication to interact and not invoke the AO subprocessor, they may do so. Objects exchanged in this way are not subject to accountable object guarantees and therefore the processes use of them is purely cooperative. The single unique aspect of cooperation which is not permitted between disjoint processes is shared space. The whole structure presented here would be greatly complicated and lose much of its implementation efficiency if it were permitted. The implementation would be constrained to maintain the shared space in the same physical memory and map the processes onto it with the inherent conflict and race problems. Processes could not be arbitrarily moved. By simply factoring processes into disjoint address space such problems do not exist. As was shown in our TENEX model, sufficient generality can be obtained within one process state to model the whole TENEX system including the shared space.

APPENDIX A
CONSTRAINTS

Postulated Design Constraints

C1. The designed network must consist of a set of interacting bounded programmable digital computer systems with well-defined processes, and be open to communication with foreign systems having undefined processes.

C2. Responsibility for meeting the postulated design constraints and design decisions must be factored between the network, implementation, and process designers.

C3. Each designer must have sufficient authority to control the effects, on the processes running in the network, of the interactions for which that designer has been delegated responsibility.

C4. Although processes must be permitted to delegate to other processes their authority to control process interactions, the delegating process always retains responsibility for that interaction and the authority to control the process to which it was delegated.

C5. The network definition must provide for modification of its definition. The design must provide sufficient constraints to be able to delegate safely to process designers all modification decisions and authority to control the effects of such decisions on the processes.

Delegation of Authority Constraints

D1. Responsibility for each interaction and authority to control it must be uniquely assigned to the same designer.

D2. Network designers are responsible for ensuring that the network is self protecting, meets the constraints, and is formally defined.

D3. The implementation designers are responsible for the design, construction, and maintenance of an optimal equivalent system.

D4. The process designers are responsible for all virtual interprocess interactions.

D5. If an implementation designer has an unsolvable problem due to boundedness which effects the processes in the network, the effect on the process state structure must be well-defined so the process designers can ameliorate the consequences.

D6. Network designers are responsible for the initial definition of the network and implicitly for all evolutionary paths. Implementation designers are responsible for the implementation of the initial definition and for any potential evolutionary path. Process designers are responsible for selecting which evolutionary paths will be followed.

Process Designer Constraints

P1. Processes must be arranged in a tree structure defining their hierarchy of responsibility over descendent processes.

P2. A unique root process must exist representing overall process management.

P3. No processor race conditions can be permitted to effect the process state transformation during that transformation.

P4. Operators available to process designers affect only the containing process state.

P5. Multiple processes may be transformed in parallel.

P6. Distribution of processes over the network must be permitted and each process state must be uniquely contained in at most one system at a time.

P7. Interprocess communication must be permitted and be subject to parental control. Synchronization between processes is the responsibility of process designers. Processes residing in different systems may be synchronized only using interprocess communication.

P8. An "accountable object" chain must exist, and processes must be able to restrict use of these accountable objects when sub-allocated.

Network Designer Constraints

N1. The network designer is responsible for providing a set of invariant names which can be used by the process designers for communication and control.

N2. The network designers are responsible for changes in the network formal definition.

N3. Sufficient services must be provided by the network designers for the process designers to accomplish their purpose, including resource accounting and intersystem communication.

N4. Network modification operators are meta-operators and must first ensure the modification does not violate any of the postulated constraints and then either act as a no-op if it is one of the permissible paths of evolution or fault if it is not.

N5. The effects of network evolution, other than the addition of a new system to the network, must be local to the system within which the operator was executed.

N6. The network design must permit process designers to limit the effects of a network modification operator to the process executing it. Process designers must be able to uniquely control the right to execute non-commuting modification operators (if any exist).

APPENDIX B
DESIGN DECISIONS

A1 - Each system processor in the network will contain a set of identifiable subprocessors including at least the following three: GP (General Programmable), AO (Accountable Object), PR (Process state Receive and transmit).

A2 - A process state is defined by a set of identifiable, typed, possibly related expression states, and associated information necessary for both subprocessor allocation and interprocess interactions.

A2.1 - Subprocessor access to structures of any other type of expression-state must not violate the conventions of the other expression-state's designer.

A3 - "Control-Points" are accountable objects.

A3.1 - Individual control-points may be paired with individual expression-states and more than one such pair may exist in a process state at one time.

A3.2 - When paired with an expression state, a control-point will have associated with it the type of subprocessor to be applied.

A4 - Control-points will serve as uniquely named destination addresses for interprocess messages. The arrival of such messages, as well as local subprocessor operations, may designate an ES/CP pair as requesting a subprocessor, regardless of the status of other pairs. The system processor will apply at most one subprocessor to a process state each process step.

A4.1 - Each process state, except the PR process state, will contain an ES/CP pair which can interrupt all others in that process and which can be interpreted by the AO subprocessor.

A4.2 - In addition to its basic functions the AO subprocessor will be responsible for regulating process birth, death, and intersystem movement.

A5 - A process may generate a single interprocess message each process step, which is broadcast by the system processor to a particular buffer associated with the destination control-point. Process designers can control the right to transmit, the right to listen, and the right to allow an ES/CP pair to become demanding as a result of a buffered message.

A6 - The initial network definition will consist of one basic system with an initial root process state. Network modification operators will permit the creation

of at least additional basic systems, subprocessors, subprocessor operators, and system processor operators for control-point housekeeping.

A6.1 - The design of modification operators will be delegated to subprocessor designers. Modification operators will not introduce new modification operators.

A7 - Inaccessible objects may result from meta-operations invoked by implementation designers because either a subprocessor hits a bound or there is an implementation failure. Subprocessors trying to access an inaccessible object will fault.

A7.1 - Implementation failures affecting the accountable object chain may cause ghost processes.

APPENDIX C

EXAMPLE FORMAL NETWORK DEFINITION

INTRODUCTION

This appendix will formally define an initial basic system and also define, as an example, a two system network. The definition system to be used here is described in FH71 and is formally defined in the appendix of Jo73. An implementation of the definition system exists on the University of Wisconsin's UNIVAC 1108. The representation used in the listing of the definitions presented later will be that accepted by the implementation in order to avoid "typographical" errors.

Although familiarity with FH71 is necessary for a precise understanding of definition details, a brief introduction to the system will be presented here so the reader who is not familiar with it will have a reasonable idea of what is being done. The definition system has the ability to represent a set of interacting asynchronous digital computer systems as extensions of Post production systems (FH71). The automaton which interprets the definition system defines the concepts of processor and process.

As in Post's formal system there is an alphabet (χ) axioms (σ:) and productions (π:). Some of the extensions to the Post system are "not x" to represent any single

character except x, where x is in the alphabet; the ability to discard irrelevant axioms; the ability of one system (SR) to communicate with another system; and the generalization of antecedents to permit the use of restricted processors (RPRs). (RPR is described in FH71 and (7) is an example of the use of an RPR in the network definition. The parenthesized numbers correspond to the parenthesized numbers found on the system listings and are used to indicate specific items which will be discussed here.)

An SR (system) has the property that it runs asynchronously with all other SR's. Each of the three systems presented in the network definition is an SR. Messages are transmitted as an axiom naming a destination axiom to be matched (in the destination SR) and a message which replaces that axiom in the destination SR. A flowchart describing the primitive automaton is found in fig. 4a, b, c. It applies the operators (productions including double arrow productions (9) which transmit messages and restricted processors (7) to the axioms, saving only generated axioms, updates the axiom set with any incoming message, and then repeats the cycle. A system step is defined as one cycle of the primitive automaton as described above.

An RPR can be a part of the left hand side of any operator. The main difference between its representation, and that of a system, is that it has a pattern, in place of an axiom set, which must be matched (using the RPR alphabet)

before the RPR is applied. RPR's operate only on the pattern matched substrings of axioms. An RPR can go through many cycles itself before returning to the containing SR, thus permitting arbitrarily complex system steps.

The flowchart (fig. 4a,b,c), symbol descriptions, and definitions were written by Pamela Smith.

SYMBOLS (fig. 4)

σ system process state set.

σ' buffer to hold generated process states during the system step in which they are generated.

σ'' message-receiving buffer.

$C(\sigma'')$ a consumable copy of σ''.

π the set of productions (operators) in the system.

$C(\pi)$ a consumable copy of π.

β a buffer for (channel, message) pairs during the system step in which they are generated.

β_P a buffer for (channel, message) pairs during the application of the production from which they are generated.

β_{RPR} a buffer for (channel, message) pairs during the execution of the RPR by which they are generated.

τ a buffer for RPR results which are generated before the RPR terminates.

P a production selected at random from $C(\pi)$.

S an element selected at random from $C(\sigma'')$; it is a message set, consisting of a unique channel name and a set of messages associated with that name.

SR a Boolean stack variable, true if an SR is being executed, false if an RPR is being executed.

ϕ the empty set.

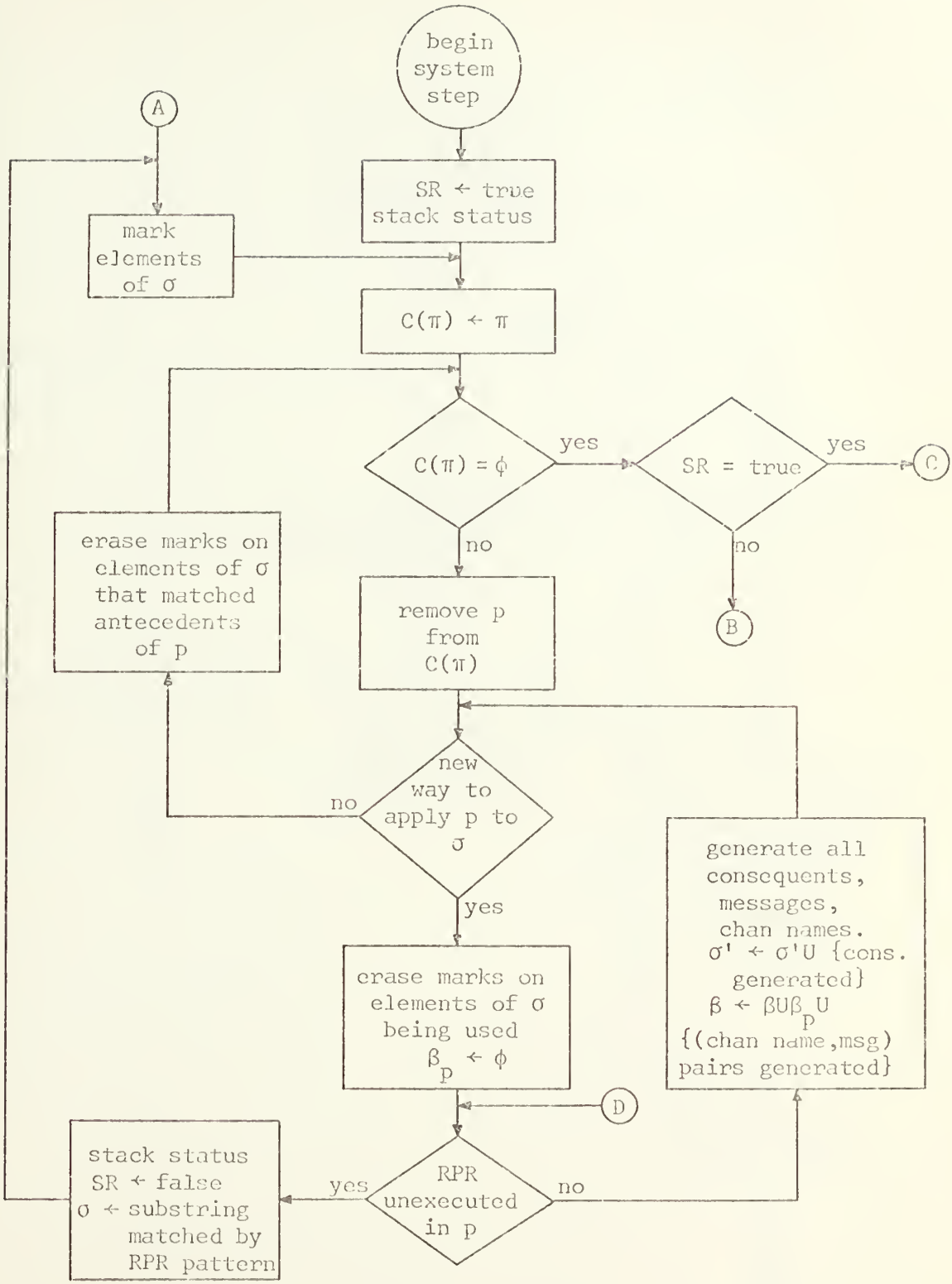


Fig. 4a--INTERPRETER Flowchart
the transformation of axioms

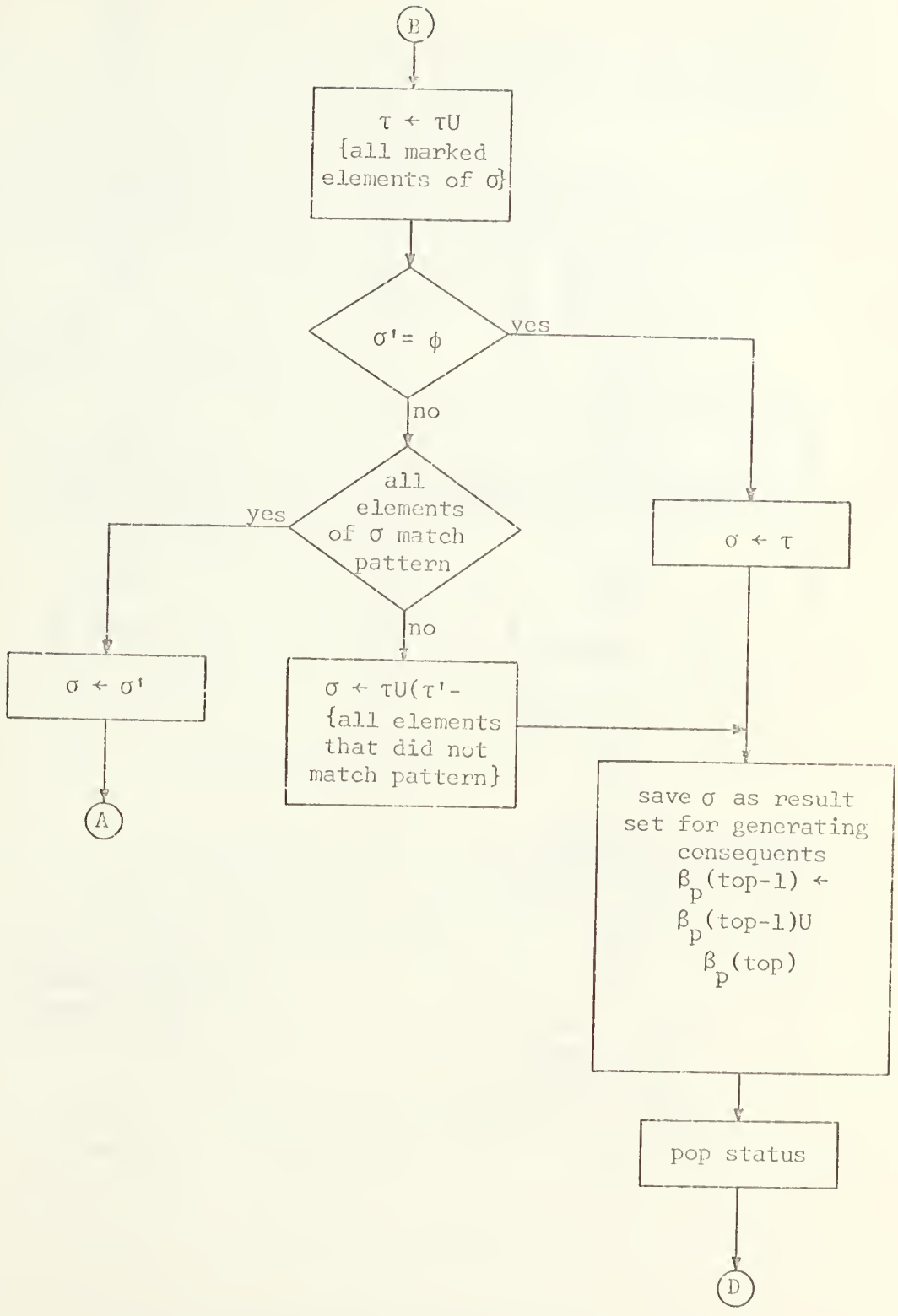


Fig. 4b--INTERPRETER Flowchart (Cont.)

termination of an RPR application

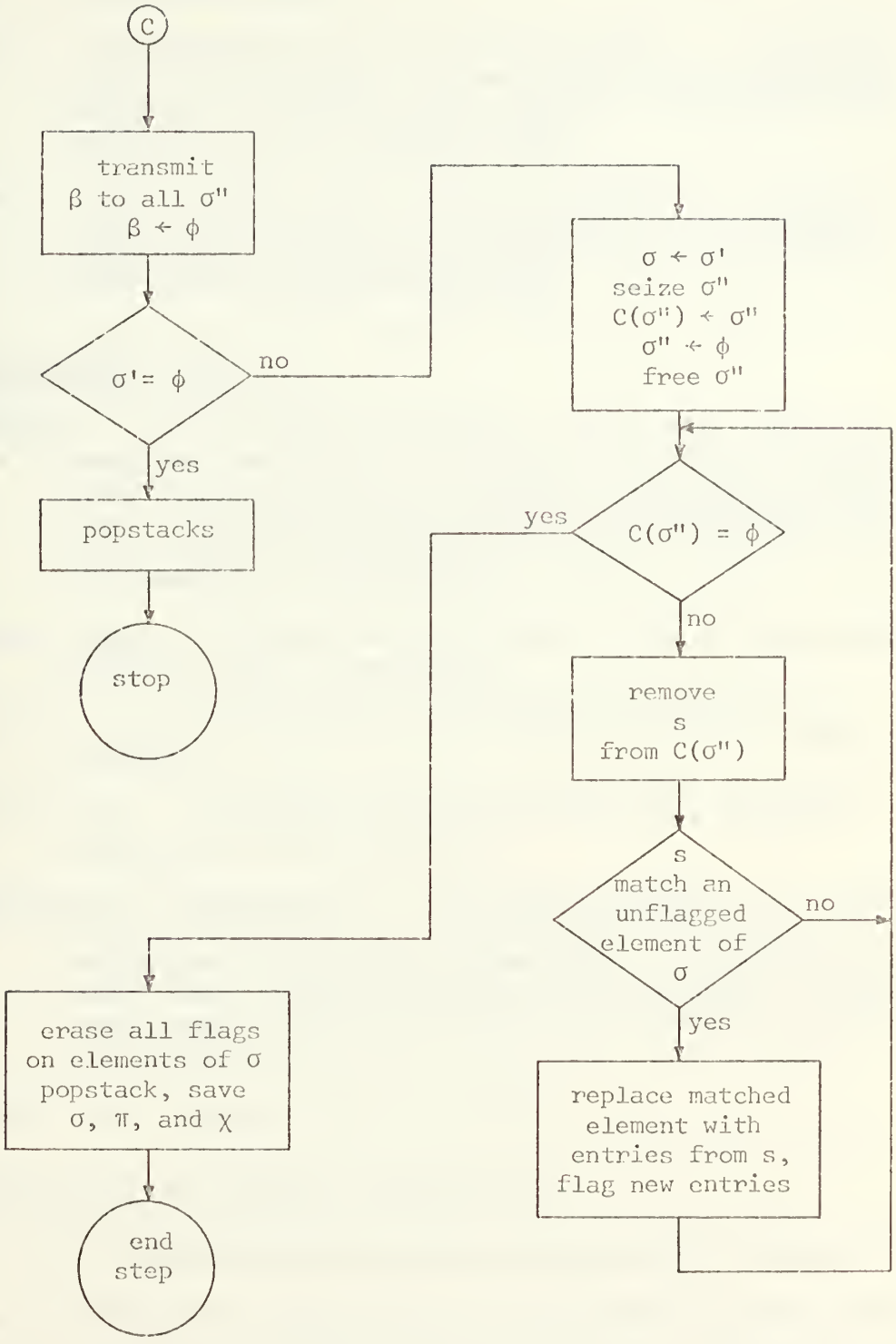


Fig. 4c--INTERPRETER Flowchart (Cont.)

SR process step termination and message management

Λ the null string.

mark a designation that can be attached to a process state, then erased; it means that no production has been applied to this process state since the beginning of the system step.

flag a designation that can be attached to a process state, then erased; it means that this process state has just been introduced to σ as a transmitted and accepted message.

DEFINITIONS (fig. 4)

TRANSMIT β The transmit primitive takes the (channel, message) pairs of β and forms them into message sets before sending. There is a conflict resolution mechanism which enables each system's σ" to be able to determine the order of arrival of the messages it receives.

SEIZE...FREE A primitive that takes σ" out of commission without loss of messages.

MATCH A $ can match any string over $(\chi+\Lambda)^*$ in any context.

APPLY P TO σ Finding a new way to do this implies binding of the variables.

GENERATE ALL CONSEQUENTS All results (states that match the pattern but no antecedents) will be used, no matter when they were generated. A result of Λ will cause the RPR pattern $ to become Λ in the consequent. No consequents will be generated if the result is ϕ.

STACK: $\sigma, \pi, \tau, \beta, \beta_p, \sigma', C(\pi)$, P, SR, χ.

DESCRIPTION OF NETWORK DEFINITIONS

In the next section three systems are defined. System A is the definition of an initial root system, and systems B and C are the two systems in a two system network example. The implementation accepts an arbitrarily large

set of "characters" in the form of alphanumeric symbols (such as Alpha or B35). Each such symbol is treated as a single character. Alphanumeric symbols may be delimited by any character (including a space) except a letter or digit.

Meta characters: The following meta characters are used by the interpreter. (Note that all of these characters can be quoted and then used just like any other character in the system definition.)

[] - beginning and end markers for an SR and RPR.

: - marks the beginning of the axiom set (σ:) of an SR (Note RPRs have only an axiom pattern).

- marks the beginning of the alphabet (χ:) of an SR or RPR.

% - marks the beginning of the productions (operators) (π:) of an SR or RPR.

& - separator between axioms in the axiom set and between antecedents in a multiple production.

; - separator between productions.

→ - arrow used in productions. A single arrow separates the antecedents and the consequent. In a double arrow production the second arrow separates the consequent message from the destination name.

\ - "NOT" (\a-matches any character in the alphabet but "a").

$ - matches any string over the alphabet including the null string.

<integer> - subscripts for variables in consequents which name variables in antecedents.

Axiom sets (1): The following is a definition of the possible state sets of the network systems (subject to the

addition of other state sets of the network as the result of network modification).

<system state>::=<system phase> <chan list> <buff list>
<process list> <candidate list>
<modify list>

phase- each system has a phase counter (3).

<system phase>::=PHASE <phase number>

<phase number>::=1|2|3

chan- each system will have a set of channels to receive messages for its control-points.

<chan list>::=<message channel>|<message channel>
<chan list>

<message channel>::=&CHAN <control-point name>
<buffer name><msg>

<control-point name>::=<AO name>|<PR name>|<GP name>

<AO name>::=<process name>AO

<process name>::=<root name>|<process name>.<child name>

<root name>::=1

<child name>::=<integer>

<integer>::=<digit>|<nzero><ndigit>

<ndigit>::=<digit>|<digit><ndigit>

<digit>::=0|<nzero>

<nzero>::=1|2|3|4|5|6|7|8|9

<PR name>::=<system number>PR

<system number>::=<integer>

<GP name>::=<priority>GP

<priority>::=<integer>

<buffer name>::=<parent buffer>|<interface buffer>|
<integer>|<acknowledge buffer>

<parent buffer>::=P|<process name>

<interface buffer>::=R

<acknowledge buffer>::=S

<msg>::=<null>|Δ<message>|Δ<inaccessible object>

<null>::=

<message>::=to be defined by subprocessor designers

<inaccessible object>::=#

buff-each system will have a set of control-point buffers.

<buff list>::=<message buffer>|<message buffer>
<buffer list>

<message buffer>::=&BUFF<control-point name><buff name>
<msg>

process-each system will have a set of processes

<process list>::=<process>|<process><process list>

<process>::=&<interface>PROCESS<process state>

<interface>::=<null>|<msg>

<process state>::=<PR state>|<user state>

<PR state>::=CP<system number>PRΔ<demand>PR[<PR exp-
state>]

<demand>::=0|1

<PR expression-state>::=<acknowledge list>

<acknowledge list>::=<ack>|<ack><acknowledge list>

<ack>::=<slist><process name>

<slist>::=S1|S2

<user state>::=CP<process name>AOΔ<demand>AO[<AO exp-
state>]

<AO exp-state>::=<AO-ES><GP-ES list>

<AO-ES>::=to be defined by AO subprocessor designer

<GP-ES list>::=null|<GP ES/CP pair>|<GP ES/CP pair>
<GP-ES list>

<GP ES/CP pair>::=<GP control-point><GP-ES>|<GP-ES>

<GP-ES>::=to be defined by the GP subprocessor designer

<GP control-point>::= CP<GP name><CP status><arm list>

<CP status>::=Δ<demand>Δ<enable>Δ<deliver order>

<enable>::=0|1

<deliver order>::=0|1

<arm list>::=<null>|<arm status>|<arm status><arm list>

<arm status>::=<status><GP name><buffer name>

<status>::=ARM|DARM

Candidate list only exists in phase 1 and tells which control-points are candidates to have a subprocessor applied.

<candidate list>::=<null>|<candidate>|<candidate> <candidate list>

<candidate>::=&<can status><control-point name>

<can status>::=CAN|NCAN

Modify list is the list of modification operations to be carried out (phase 3 only).

<modify list>::=<null>|<modify>|<modify><modify list>

<modify>::=&MODIFY<process name><mod><GP name>

<mod>::=NEW|KILL

State set restrictions: the following constraints will hold on the state set of each system in the network.

a) The system will contain one process with a <PR state> with the proper <system number>. There will be one <message channel> with corresponding identification for

each other system in the network and one PR <acknowledge buffer>.

b) The system will contain a finite number of processes with <user state>.

c) For each <AO name>, one <message channel> will exist for <parent buffer>, <interface buffer>, and <acknowledge buffer> along with one <message channel> (with appropriate identification) for each of that processes' children.

d) For each <CP name><buffer name> in an <arm status> there will be a <message channel> with the corresponding <CP name><buffer name>.

e) For each <CP name><buffer name> in an <arm status> there will be one <message buffer> with the corresponding <CP name><buffer name>.

f) On Phase 1 there will be one <candidate> for each <control-point name>.

We would consider an appropriate path of evolution, as a result of modification operator execution, any system which would meet these constraints.

System processor operators: The % production set (3) of each system defines its operators. The line numbers refer to the root system definition as a specific example.

Lines 9-11 (3): These operators maintain the phase counter (Fig. 5).

Line 12: Print out axiom set for system verification purposes. This operator is not required as part of the system processor operation.

Lines 13-26 (4): GP message buffer operations are carried out during all three phases as follows. Line 13 clears all GP input channels at each phase. Lines 24, 25 remember any message already in a GP buffer (if buffer is still armed), if no new message has arrived on the corresponding input channel. Line 26 buffers any new message (if the buffer is armed). Note that the productions, lines 24 and 26, cause only the most recent message to be buffered. Line 14 ensures that any disarmed buffer is empty at the end of phase 3, the other productions are applied in each phase. There will be a set of productions similar to lines 24-26 for each GP buffer.

Lines 16-23: AO and PR message operations are different in phase 3 than in phase 1 and phase 2. Lines 17, 18, 22, and 23 update AO buffers in phase 3, and line 19 updates the PR buffers in phase 3. Line 16 remembers AO and PR channel messages in phase 2 and phase 3. No buffers are required for AO or PR control-points except during phase 1 since their communication is, by design, cooperative. There will be one production like each of lines 22 and 23 for each AO in the system.

Phase 1: Designate which control-points are to have subprocessors applied (Fig. 5) (5). The control-point

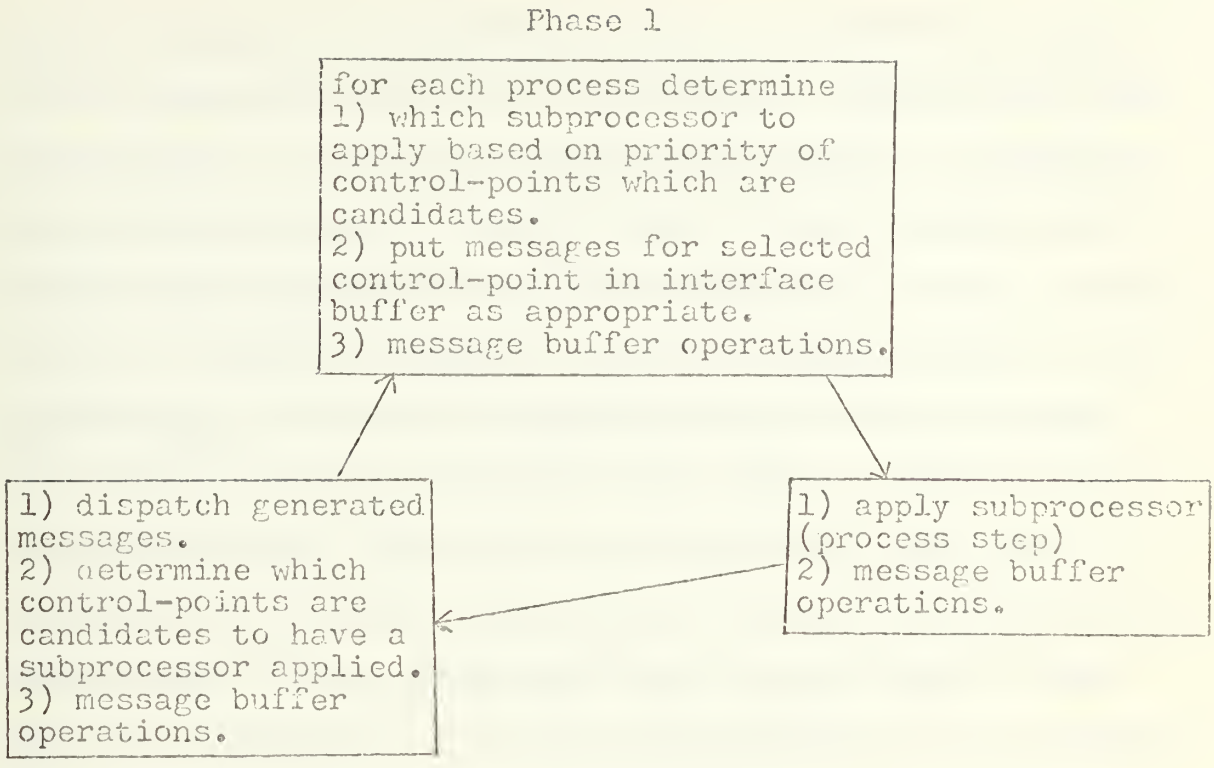


Fig. 5 -- System Phases: the system processor has a cycle of three system steps which includes one process step (phase 2).

designated within each process will be indicated by a ":". A process which has no control-points which are to have a subprocessor applied will be marked with a ":" preceding its state (lines 32 and 48). If an AO is demanding (line 27, 28) it automatically gets a subprocessor applied since, by design, it is always the highest priority control-point in a process. If AO has been designated a candidate (done in phase 3 as a result of being demanding or having a pending message), and if it is not currently demanding, then lines

33-35 designate it demanding, place all buffered messages in the interface buffer for that process, and mark the process (with a :) to have a subprocessor applied. Lines 36-46 determine which GP control-point (if any) should have a subprocessor applied if the AO control-point in that process is not a candidate (NCAN). For any GP control-point to be designated to receive the subprocessor, it must be "CAN", and "NCAN" must hold for all higher priority control-points in that process. Then, if it is demanding (lines 36, 37), designate it to have a subprocessor applied (no messages are delivered). If not demanding, place either single (lines 38, 39) or multiple (lines 42, 43) messages in the interface buffer, and designate the control-points to have a subprocessor applied. Lines 40, 41 and 44, 45, 46 clear the buffers when messages are put into the interface buffer.

Lines 27-32 designate PR to receive a subprocessor. During phase 1, PR gets all pending messages put in its interface buffer (lines 29, 30). If there are no messages, and it is demanding, then line 31 designates it to have a subprocessor applied. There will be a set of operations like lines 33-47 for each process (except PR) in that system.

Subprocessors are applied during Phase 2(6). If the process was marked as having no control-point which was to have a subprocessor applied, then the mark is removed (line 48). The AO subprocessor (lines 83-87) and GP sub-

processor (lines 88-92) are not part of this design. As defined here they only clear the interface buffer and set the control-point not demanding (lines 85 and 90), or if there is no message in the interface buffer, they set the control-point not demanding (lines 86 and 91).

The PR subprocessor (lines 51-82) (7) has been defined as follows. PR may get multiple messages in its interface buffer (! is used as a message separator). Lines 54, 55 execute a modification operator for incoming requests to create a new process in the local system. The modification request tells the system processor which control-points a process contains. Line 56 turns off the demand bit (if there is no message in the interface buffer (! Process) and if the PR expression-state has no work to do ('[S1$])) and removes the "!" which will cause the subprocessor to terminate. Lines 55-58 move an incoming message to the expression-state for later FIFO service. Line 59 clears the null messages which are delivered for empty buffers. Line 60-62 removes one local request (to create a process) from the expression-state, transforms it, and places it in the interface buffer as a "start" request. Lines 63-66 send back to a source system an acknowledgment that the process state arrived correctly and marks the state to be started the next time the PR subprocessor is applied. Lines 67-69 start such a process. Lines 70-71 send back to a source system over the acknowledgment channel that an inaccessible

object (#) arrived in place of the process state. Lines 72-74 do not transmit another process state since there is already one outstanding (S1↖Δ$']). Lines 75-78 transmit a process state when there is no transmission pending acknowledgments. Lines 80, 81 notify the proper AO by an "S" if the move was successful, and by an "R" if it was not. If a PR receives a returned inaccessible object it will notify the AO control-point that initiated the move that the request failed. If a PR receives a process state it will return to the sending PR an acknowledgment. The PR originally sending the process state will notify the AO which requested the move that it succeeded. That AO will then kill that process state since it now exists in the other system. This procedure guarantees that no process state will be logically lost during intersystem movement.

Phase 3 (8) dispatches messages and designates potential control-point candidates for subprocessor application. Line 93 and 95 designate a demanding AO or GP as a candidate (CAN). All non-demanding AO and GP control-points are designated not candidates (NCAN) (lines 94 and 97). NCAN can be reset (by message overwrite) as a result of a message on a channel (line 102 and 104) provided the control-point is enabled and the buffer armed (AO is always enabled and all AO buffers are always armed). Messages generated during phase 2 which are for local control-points, also reset the control-point to CAN (line 109). The candi-

date mechanism permits us to know in advance which control-points are candidates to have a subprocessor applied.

Lines 106-137 do message dispatching.

Lines 106 and 113 automatically buffer locally any message generated for a local CP if it is armed.

Lines 111, 121, 124 dispatch any non-local message.

Line 113 kills a process and executes appropriate modification operations.

Line 127 starts a process (request was from PR).

Line 130 buffers a move request for PR.

Lines 136, 137 create empty PR buffers.

SYSTEM A

```
 1 (1)  [: PHASE 3 & CHAN 1 AO P & CHAN 1 AO R &
 2         CHAN 1 AO S &
 3         CHAN 1 GP 1 & BUFF 1 GP 1
 4         & PROCESS CP 1 PR^1 PR'[S1']
 5         & PROCESS CP 1 AO^1 AO'[' ' CP 1 GP^1^1^0 ARM 1 GP 1 GP'[']']
 6 (2)   # PHASE 0 1 2 3 4 CHAN BUFF PROCESS ARM DARM CAN
 7         NCAN  MODIFY KILL PR AO GP CP P R S ^ '[ '] '  ': ! .
 8         MOVE START NEW TEST '# S1 S2
 9 (3)   % PHASE 1 -> PHASE 2 ;
10         PHASE 2 -> PHASE 3 ;
11         PHASE 3 -> PHASE 1 ;
12         $ -> @ S1' ' $<1> -> PRINT 1 ;
13 (4)     CHAN\^GP\^$ -> CHAN\^<1>GP\^<2> ;
14         PHASE 3 & $PROCESS CP[$#1 2 .%]AO $DARM\^\^\^$
15             -> BUFF\^<1>\^<2>\^<3> ;
16         PHASE \3 & CHAN[$#1 2.%][\GP#AO PR%]$ -> CHAN$<1>\GP<1>$<2> ;
17         PHASE 3 & $PROCESS CP 1 AO^1$ & CHAN 1 AO$ -> CHAN 1 AO$<3> ;
18         PHASE 3 & ^MOVE\^CP 1 AO$ & CHAN 1 AO$ -> CHAN 1 AO$<2>;
19         PHASE 3 & CHAN\^PR\^^$ -> BUFF\^<1>PR\^<2>^\^<2>$<1> ;
20         PHASE 3 & CHAN \^PR\^ -> BUFF\^<1>PR\^<2>^ ;
21         PHASE 3 & CHAN \^ PR\^$ -> CHAN \^<1> PR\^<2> ;
22         PHASE 3 & $PROCESS CP 1 AO^0$ & CHAN 1 AO$ -> BUFF 1 AO$<3> ;
23         PHASE 3 & $PROCESS CP 1 AO^0$ & CHAN 1 AO\^$ -> CHAN 1 AO\^<1> ;
24         CHAN 1 GP 1 & BUFF 1 GP 1$ & $PROCESS$ARM 1 GP 1$
25           -> BUFF 1 GP 1$<1> ;
26         CHAN 1 GP 1^$ & $ARM 1 GP 1$ -> BUFF 1 GP 1^$<1> ;
27 (5)     PHASE 1 & PROCESS CP[$#1 2 .%]AO^1$
28              -> PROCESS CP$<1>':AO^1$<2> ;
29         CAN 1 PR & PROCESS CP 1 PR^\^$ & BUFF 1 PR 1^$
30           -> !$<2>!PROCESS CP 1';PR^1$<1> ;
31        NCAN 1 PR & PROCESS CP 1 PR^1$ -> !PROCESS CP 1 ':PR^1$<1> ;
32        NCAN 1 PR & PROCESS CP 1 PR^0$ -> ';PROCESS CP 1 PR^0$<1> ;
33         CAN 1 AO & PROCESS CP 1 AO^0$ & BUFF 1 AO P$ & BUFF 1 AO R$
34           & BUFF 1 AO S$
35                -> $<2>$<3>$<4>PROCESS CP 1';AO^1$<1> ;
36         NCAN 1 AO & CAN 1 GP & $CP 1 GP^1$
37                -> $<1>CP 1';GP^1$<2> ;
38         NCAN 1 AO & CAN 1 GP & $CP 1 GP^0^1^0$
39                & BUFF 1 GP 1^$ -> ^$<3>$<1>CP 1';GP^1^1^0$<2> ;
40         NCAN 1 AO & CAN 1 GP & $CP 1 GP^0^1^0$ & CHAN 1 GP 1$
41                & BUFF 1 GP 1^$ -> BUFF 1 GP 1$<3> -> BUFF 1 GP 1^$<4> ;
42         NCAN 1 AO & CAN 1 GP & $CP 1 GP^0^1^1$
43                & BUFF 1 GP 1$ -> $<3>$<1>CP 1';GP^1^1^1$<2> ;
44         NCAN 1 AO & CAN 1 GP & $CP 1 GP^0^1^1$ & CHAN 1 GP 1$
45                & BUFF 1 GP1$
46                -> BUFF 1 GP 1$<3> -> BUFF 1 GP 1$<4> ;
47         NCAN 1 AO & NCAN 1 GP & PROCESS CP 1 AO$ -> ';PROCESS CP 1 AO$<1>;
48 (6)     PHASE 2 & ':PROCESS$ -> PROCESS$<1> ;
49         PHASE 3 & $PROCESS$ -> PROCESS$<2> ;
50         PHASE 2 -> MODIFY 1;
51 (7)     PHASE 2 & [$':PR$#PROCESS ARM DARM CP P R S FR AO GP ^ 0
52                   1 2 3 4 '[ '] '  START MOVE NEW TEST : . CHAN
53                   '# ': S1 S2
54           %!\^NEW\^CP[$#1 2.%]AO$CP[\^#1 2 3 4%]GP$':$
55                   -> MODIFY$<1>NEW\^<3>GP -> MODIFY 1 ;
56            !PROCESS$PR^1 PR'[S1$ -> PROCESS$<1>PR^0 PR'[S1$<2> ;
```

```
57              !C\^$#ARM OARM CP P R S AO GP ^ O 1 2 3 4 'C ']  . '# S1 S2
58                 '  TEST MOVE NEW&]!$';$S1$ -> !$<2>';$<3>\^<1>$<1>!S1$<4> ;
59              !!S';$ -> !$<1>';$<2> ;
60              !PROCESS$PR'CO NEW OC$#ARM DARM CP P R S AO GP CP PR ^ .
61                    O 1 2 3 4 'C ']  '  TEST MOVE NEW '# S1 S2
62                 &]!$ -> ^START$<2>PROCESS$<1>PR'C$<3> ;
63              !PROCESS$ PR'C\O NE#\OC$#ARM DARM CP P R S
64                   AO GP ^ O 1 2 3 4 'C ']  '  .&]!$
65               -> ^CHAN\O<1>PR S^R PROCESS$<1>
66                    PR'CSTART$<2>!$<3> ;
67              !PROCESS$PR'CSTARTC$#ARM DARM CP P R S AO GP ^
68                   O 1 2 3 4 'C ']  '  . S1 S2&]!$
69               ->^START$<2>PROCESS$<1>PR'C$<3> ;
70              !PROCESS$PR'C\^'#!$
71               -> ^CHAN\^<1>PR S^S PROCESS$<1>PR'C$<2>;
72              !PROCESS$PR'CO MOVE\OC$#ARM DARM CP P R S
73                AO GP ^ O 1 2 3 4 'C ']  '  .&]!$S1\^$']
74               ->PROCESS$<1>PR'C$<3>O MOVE\O<1>$<2>!S;\^<1>$<4>'] ;
75              !PROCESS$PR'CO MOVE\OCPC$#1 2 .&]AOC$
76                   # ARM DARM CP P R S AO GP ^ O 1 2 3 4
77                     'C ']  '  . &]!$S1']
78                 ->^CHAN\O<1>PR 1^NEW 1 CP$<2>AO$<3>
79                   PROCESS$<1>PR'C$<4>S1$<2>'] ;
80              !PROCESS$PR'CSC\^#R S&]!$S1$']
81                ->^CHAN$<]>AO R^\^<1>PROCESS$<1>PR'C$<2>S1'] ;
82                   ] -> $<1>PR$<2> ;
83           PHASE 2 & C$';AO$#PROCESS ARM DARM P R S CP AO GP ^ O 1
84                     2 3 4 'C ']  '  '# . TEST
85                 &^$PROCESS CPC$#1 2 .&]';AO^1$ -> PROCESS CP$<2>';AO^1$<3> ;
86                 PROCESS CPC$#1 2 .&]';AO^1$ -> PROCESS CP$<1>';AO^O$<2> ;
87                   ] -> $<1>AO$<2> ;
88           PHASE 2 & C$';GP$ #PROCESS ARM DARM P R S AO GP ^ O 1
89                     TEST CP 2 3 4 'C ']  '  '# .
90                     & ^$PROCESS$';GP^1$ -> PROCESS$<2>';GP^1$<3> ;
91                     PROCESS$';GP^1$ -> PROCESS$<1>';GP^O$<2> ;
92                   ] -> $<1>GP$<2> ;
93   (8)     PHASE 3 & $PROCESS CPC$#1 2 .&]AO^1$ -> CAN $<2>AO ;
94           PHASE 3 & $PROCESS CPC$#1 2 .&]AO^O$ -> NCAN $<2> AO ;
95           PHASE 3 & $PROCESS CPC$#1 2 .&]AO$CPC\^#1 2 3 4&]GP^1$
96               ->CAN \^<1>GP ;
97           PHASE 3 & $PROCESS CPC$#1 2 .&]AO$CPC\^#1 2 3 4&]GP^O$
98               -> NCAN \^<1> GP ;
99           PHASE 3 & ^MOVE$CPC$#1 2 3 4.&]C\^#AO GP&]$
100            -> NCAN$<2>\^<1> ;
101          PHASE 3 -> NCAN 1 PR ;
102          PHASE 3 & CHANC$#1 2 .&]C\GP#AO PR&]\^^$
103              -> CAN$<1>\GP<1> -> NCAN$<1>\GP<1> ;
104          PHASE 3 & C\^#CHAN BUFF&] 1 GP 1^5 & $CP 1 GP^O^1$ARM 1 GP 1$
105               -> CAN 1 GP -> NCAN 1 GP ;
106          PHASE 3 & ^CHAN 1 GP 1^$PROCESS$ & $CP 1 GP^O^1$ARM 1 GP 1$
107             & C\^#CHAN BUFF&]1 GP 1$
108             -> BUFF 1 GP 1^$<1> -> BUFF ! GP 1$<6> ;
109          PHASE 3 & ^CHAN 1 GP 1^$PROCESS$ & $CP 1 GP^O^1$ARM 1 GP 1$
110               -> CAN 1 GP -> NCAN 1 GP ;
111  (9)     PHASE 3 & ^CHANC\^#2 3 4&]GP\^$PROCESS$
112               -> CHAN\^<1>GP\^<2>$<1> -> CHAN\^<1>GP\^<2> ;
113          PHASE 3 & KILLC$#'C ']  '  ARM DARM CP P R S AO GP ^ .
```

```
114                          0 1 2 3 4 '#
115                        $$CP[\^\^#1 2 3 4 AO GP$]$ ->
116                            MODIFY\^<1>\^<2>KILL -> MODIFY 1 ;
117                             ] -> ;
118          PHASE 3 & ^CHAN[$#1 2.$]AO$PROCESS$ -> BUFF$<1>AO$<2> ;
119          PHASE 3 & $PROCESS CP 1 AO^0$ & ^CHAN 1 AO R$
120            -> CAN 1 AO -> NCAN 1 AO ;
121          PHASE 3 & $PROCESS CP 1 AO^1$ & ^CHAN 1 AO R$PROCESS$
122            & CHAN 1 AO R$
123            -> CHAN 1 AO R$<5>$<3> -> CHAN 1 AO R$<5> ;
124          PHASE 3 & ^CHAN[$AO\^#1 2 3 4 .
125                           $ 1 AO\^ -> . AO . ;
126             ]$PROCESS$ -> CHAN$<1>AO\^<1>$<2> -> CHAN$<1>AO\^<1>;
127          PHASE 3 & ^START$PROCESS CP 1 PR$ -> PROCESS$<1> ;
128          PHASE 3 & ^CHAN[\^#2$]PR\^$PROCESS$
129              ->CHAN\^<1>PR\^<2>$<1> -> CHAN\^<1>PR\^<2> ;
130          PHASE 3 & ^MOVE\^CP[$#1 2 .$]AO$
131              -> BUFF 1 PR$<1>^0 MOVE\^<1>CP$<1>AO$<2> ;
132            PHASE 3 & ^MOVE\^$ -> PROCESS $<1> ;
133          PHASE 3 & ^[\^#MOVE NEW$]$ -> CAN 1 PR -> NCAN 1 PR ;
134          PHASE 3 & ^NEW$PROCESS CP[$#1 2 .$]AO$ -> BUFF 1 PR$<2>
135                ^0 NEW O$<2> ;
136          PHASE 3 & ^CHAN$PROCESS CP[$#1 2 .$]AO$ -> BUFF 1 PR$<2>^ ;
137          PHASE 3 & PROCESS CP[$#1 2 .$]AO$ ->^ BUFF 1 PR$<1> ;
138       ]
```

SYSTEM B

```
 1  (1) [: PHASE 3 & CHAN 1 AO P & CHAN 1 AO R
 2         & CHAN 1 AO 1 & CHAN 1 AO 2
 3         & CHAN 1 AO S & CHAN 1 PR S
 4         & CHAN 1 PR 2
 5         & CHAN 1 GP 1 & BUFF 1 GP 1
 6         & CHAN 1 GP 2 & BUFF 1 GP 2
 7         & CHAN 1 GP 3 & BUFF 1 GP 3
 8         & MODIFY 1
 9             & PROCESS CP 1 PR^1 PR'[S1']
10         & PROCESS CP 1 AO^0 AO'[' ' CP 1 GP^0^1^0 DARM
11            1 GP 1 DARM 1 GP 2 DARM 1 GP 3 GP'[']']
12  (2)  # PHASE 0 1 2 3 4 CHAN BUFF PROCESS ARM DARM CAN
13         NCAN  MODIFY KILL PR AO GP CP P R S ^ '[ ']  '  ': ! .
14         MOVE START NEW TEST '# S1 S2
15  (3)  $ PHASE 1 -> PHASE 2 ;
16         PHASE 2 -> PHASE 3 ;
17         PHASE 3 -> PHASE 1 ;
18         $ -> ; S1' ' $<1> -> PRINT 1 ;
19  (4)    CHAN\^GP\^$ -> CHAN\^<1>GP\^<2> ;
20         PHASE 3 & $PROCESS CP[$#1 2 .$]AO $DARM\^\^\^$
21             -> BUFF\^<1>\^<2>\^<3> ;
22         PHASE \3 & CHAN[$#1 2.$][\GP#AO PR$]$ -> CHAN$<1>\GP<1>$<2> ;
23         PHASE 3 & $PROCESS CP 1 AO^1$ & CHAN 1 AO$ -> CHAN 1 AO$<3> ;
24         PHASE 3 & ^MOVE\^CP 1 AO$ & CHAN ! AO $
25           -> CHAN 1 AO$<2> ;
26         PHASE 3 & CHAN\^PR\^^$ -> BUFF\^<1>PR\^<2>^\^<2>$<1> ;
27         PHASE 3 & CHAN\^PR\^ -> BUFF \^<1>PR\^<2>^ ;
28         PHASE 3 & CHAN \^ PR\^$ -> CHAN \^<1> PR\^<2> ;
29         PHASE 3 & $PROCESS CP 1 AO^0$ & CHAN 1 AO$ -> BUFF 1 AO$<3> ;
30         PHASE 3 & $PROCESS CP 1 AO^0$ & CHAN 1 AO\^$ -> CHAN 1 AO\^<1> ;
31         CHAN 1 GP ! & BUFF 1 GP 1$ & $PROCESS$ARM 1 GP 1$
32           -> BUFF 1 GP 1$<1> ;
33         CHAN 1 GP 2 & BUFF 1 GP 2$ & $PROCESS$ARM 1 GP 2$
34           -> BUFF 1 GP 2$<1> ;
35         CHAN 1 GP 3 & BUFF 1 GP 3$ & $PROCESS$ARM 1 GP 3$
36           -> BUFF 1 GP 3$<1> ;
37         CHAN 1 GP 1^$ & $PROCESS$ARM 1 GP 1$ -> BUFF 1 GP 1^$<1> ;
38         CHAN 1 GP 2^$ & $PROCESS$ARM 1 GP 2$ -> BUFF 1 GP 2^$<1> ;
39         CHAN 1 GP 3^$ & $PROCESS$ARM 1 GP 3$ -> BUFF 1 GP 3^$<1> ;
40  (5)    PHASE 1 & PROCESS CP[$#1 2 .$]AO^1$
41              -> PROCESS CP$<1>':AO^1$<2> ;
42         CAN 1 PR & PROCESS CP 1 PR^\^$ & BUFF 1 PR !^$
43           & BUFF 1 PR 2^$ & BUFF 1 PR S^$
44           -> !$<2>!$<3>!$<4>!PROCESS CP 1':PR^1$<1> ;
45        NCAN 1 PR & PROCESS CP 1 PR^1$ -> !PROCESS CP 1 ':PR^1$<1> ;
46        NCAN 1 PR & PROCESS CP 1 PR^0$ -> ':PROCESS CP 1 PR^0$<1> ;
47         CAN 1 AO & PROCESS CP 1 AO^0$ & BUFF 1 AO P$
48           & BUFF 1 AO R$ & BUFF 1 AO S$
49           & BUFF 1 AO 1$ & BUFF 1 AO 2$
50               -> $<2>$<3>$<4>$<5>$<6>PROCESS CP 1':AO^1$<1> ;
51         NCAN 1 AO & CAN 1 GP & $CP 1 GP^1$
52               -> $<1>CP 1':GP^1$<2> ;
53         NCAN 1 AO & CAN 1 GP & $CP 1 GP^0^1^0$
54               & BUFF 1 GP 1^$ -> ^$<3>$<1>CP 1':GP^1^1^0$<2> ;
55         NCAN 1 AO & CAN 1 GP & $CP 1 GP^0^1^0$ & CHAN 1 GP 1$
56               & BUFF 1 GP 1^$ -> BUFF 1 GP 1$<3> -> BUFF 1 GP 1^$<4> ;
```

```
57          NCAN 1 AO & CAN 1 GP & $CP 1 GP^0^1^0$
58            & BUFF 1 GP 1 & BUFF 1 GP 2^$
59            -> ^$<3>$<1>CP 1';GP^1^1^0$<2> ;
60          NCAN 1 AO & CAN 1 GP & $CP 1 GP^0^1^0$
61            & BUFF 1 GP 1 & BUFF 1 GP 2^$ & CHAN 1 GP 2$
62            ->BUFF 1 GP 2$<4> -> BUFF 1 GP 2 ^$<3> ;
63          NCAN 1 AO & CAN 1 GP & $CP 1 GP^0^1^0$
64            & BUFF 1 GP 1 & BUFF 1 GP 2 & BUFF 1 GP 3^$
65            -> ^$<3>$<1>CP 1';GP^1^1^0$<2> ;
66          NCAN 1 AO & CAN 1 GP & $CP 1 GP^0^1^0$
67             & BUFF 1 GP 1 & BUFF 1 GP 2 & CHAN 1 GP 3$
68            &BUFF 1 GP 3^$ -> BUFF 1 GP 3$<3> -> BUFF 1 GP 3^$<4> ;
69          NCAN 1 AO & CAN 1 GP & $CP 1 GP^0^1^1$
70            & BUFF 1 GP 1$ & BUFF 1 GP 2$ & BUFF 1 GP 3$
71                ->$<3>$<4>$<5>$<1>CP 1';GP^1^1^1$<2> ;
72          NCAN 1 AO & CAN 1 GP & $CP 1 GP^0^1^1$ & CHAN 1 GP 1$
73                & BUFF 1 GP 1$
74                -> BUFF 1 GP 1$<3> -> BUFF 1 GP 1$<4> ;
75          NCAN 1 AO & CAN 1 GP & $CP 1 GP^0^1^1$
76            &CHAN 1 GP 2$ & BUFF 1 GP 2$
77            -> BUFF 1 GP 2 $<3> -> BUFF 1 GP 2$<4> ;
78          NCAN 1 AO & CAN 1 GP & $CP 1 GP^0^1^1$
79            & CHAN 1 GP 3$ & BUFF 1 GP 3$
80            -> BUFF 1 GP 3$<3> -> BUFF 1 GP 3$<4> ;
81          NCAN 1 AO & NCAN 1 GP & PROCESS CP 1 AO$ -> ':PROCESS CP 1 AO$<1>;
82    (6)   PHASE 2 & ':PROCESS$ -> PROCESS$<1> ;
83          PHASE 3 & $PROCESS$ -> PROCESS$<2> ;
84          PHASE 2 -> MODIFY 1;
85    (7)   PHASE 2 & C$':PR$#PROCESS ARM DARM CP P R S PR AO GP ^ O
86                1 2 3 4 'C 'J '  START MOVE NEW TEST ! . CHAN
87                '# ': S1 S2
88            %!\^NEW\^CPC$#1 2.%]AO$CPC\^#1 2 3 4%]GP$':$
89                  -> MODIFY$<1>NEW\^<3>GP -> MODIFY 1 ;
90             !PROCESS$PR^1 PR'CS1$ -> PROCESS$<1>PR^0 PR'LS1$<2> ;
91             !C\^$#ARM DARM CP P R S AO GP ^ 0 1 2 3 4 'C 'J . '# S1 S2
92                '  TEST MOVE NEW%]!$':$S1$ -> !$<2>';$<3>\^<1>$#1>!S1$<4> ;
93             !!$':$ -> !$<1>':$<2> ;
94             !PROCESS$PR'CO NEW OC$#ARM DARM CP P R S AO GP CP PR ^ .
95                   0 1 2 3 4 'C 'J '  TEST MOVE NEW '# S1 S2
96                %]!$ -> ^START$<2>PROCESS$<1>PR'C$<3> ;
97             !PROCESS$ PR'C\O NEW\OC$#ARM OARM CP P R S
98                  AO GP ^ 0 1 2 3 4 'C 'J '  .%]!$
99              -> ^CHAN\O<1>PR S^R PROCESS$<1>
100                   PR'CSTART$<2>!$<3> ;
101            !PROCESS$PR'CSTARTC$#ARM DARM CP P R S AO GP ^
102                 0 1 2 3 4 'C 'J '  . S1 S2%]!$
103             ->^START$<2>PROCESS$<1>PR'C$<3> ;
104            !PROCESS$PR'C\^'#!$
105             -> ^CHAN\^<1>PR S^S PROCESS$<1>PR'C$<2>;
106            !PROCESS$PR'CO MOVE\OC$#ARM DARM CP P R S
107              AO GP ^ 0 1 2 3 4 'C 'J '  .%]!$S1\^$']
108             ->PROCESS$<1>PR'C$<3>O MOVE\O<1>$<2>!S1\^<1>$<4>'] ;
109            !PROCESS$PR'CO MOVE\OCPC$#1 2 .%]AOC$
110                 # ARM DARM CP P R S AO GP ^ 0 1 2 3 4
111                   'C 'J '  . %]!$S1']
112               ->^CHAN\O<1>PR 1^NEW 1 CP$<2>AO$<3>
113                 PROCESS$<1>PR'C$<4>S1$<2>'] ;
```

```
114             !PROCESS$PR'[S[\^#R S&]!$S1$']
115               ->^CHAN$<3>AO R^\^<1>PROCESS$<1>PR'[$<2>S1'] ;
116                  ] -> $<1>PR$<2> ;
117           PHASE 2 & [$';AO$#PROCESS ARM DARM P R S CP AO GP ^ 0 1
118                     2 3 4 '[ '] '  ': '# . TEST
119                 &^$PROCESS CP[$#1 2 .&]';AO^1$ -> PROCESS CP$<2>';AO^1$<3> ;
120                 PROCESS CP[$#1 2 .&]';AO^1$ -> PROCESS CP$<1>';AO^0$<2> ;
121                  ] -> $<1>AO$<2> ;
122           PHASE 2 & [$';GP$ #PROCESS ARM DARM P R S AO GP ^ 0 1
123                     TEST CP 2 3 4 '[ '] '  '# .
124                     & ^$PROCESS$';GP^1$ -> PROCESS$<2>';GP^1$<3> ;
125                     PROCESS$';GP^1$ -> PROCESS$<1>';GP^0$<2> ;
126                  ] -> $<1>GP$<2> ;
127   (8)     PHASE 3 & $PROCESS CP[$#1 2 .&]AO^1$ -> CAN $<2>AO ;
128           PHASE 3 & $PROCESS CP[$#1 2 .&]AO^0$ -> NCAN $<2> AO ;
129           PHASE 3 & $PROCESS CP[$#1 2 .&]AO$CP[\^#1 2 3 4&]GP^1$
130               ->CAN \^<1>GP ;
131           PHASE 3 & $PROCESS CP[$#1 2 .&]AO$CP[\^#1 2 3 4$]GP^0$
132               -> NCAN \^<1> GP ;
133           PHASE 3 & ^MOVE$CP[$#1 2 3 4.&][\^#AO GP&]$
134             -> NCAN$<2>\^<1> ;
135           PHASE 3 -> NCAN 1 PR ;
136           PHASE 3 & CHAN[$#1 2 .&][\GP#AO PR&]\^^$
137               -> CAN$<1>\GP<1> -> NCAN$<1>\GP<1> ;
138           PHASE 3 & [\^#CHAN BUFF&] 1 GP 1^$ & $CP 1 GP^0^1$ARM 1 GP 1$
139                 -> CAN 1 GP -> NCAN 1 GP ;
140           PHASE 3 & [\^#CHAN BUFF&]1 GP 2^$ & $CP 1 GP^0^1$ARM 1 GP 2$
141             -> CAN 1 GP -> NCAN 1 GP ;
142           PHASE 3 & [\^#CHAN BUFF&]1 GP 3^$ & $CP 1 GP^0^1$ARM 1 GP 3$
143             -> CAN 1 GP -> NCAN 1 GP ;
144           PHASE 3 & ^CHAN 1 GP 1^$PROCESS$ & $CP 1 GP^0^1$ARM 1 GP 1$
145              & [\^#CHAN BUFF&]1 GP 1$
146              -> BUFF 1 GP 1^$<1> -> BUFF 1 GP 1$<6> ;
147           PHASE 3 & ^CHAN 1 GP 2^$PROCESS$ & $CP 1 GP^0^1$ARM 1 GP 2$
148             & [\^#CHAN BUFF&]1 GP 2$
149             -> BUFF 1 GP 2^$<1> -> BUFF 1 GP 2$<6> ;
150           PHASE  3& ^CHAN 1 GP3^$PROCESS$ & $CP 1 GP^0^1$ARM 1 GP 3$
151             & [\^#CHAN BUFF&]1 GP 3$
152             -> BUFF 1 GP 3^$<1> -> BUFF 1 GP 3$<6> ;
153           PHASE 3 & ^CHAN 1 GP 1^$PROCESS$ & $CP 1 GP^0^1$ARM 1 GP 1$
154                 -> CAN 1 GP -> NCAN 1 GP ;
155           PHASE 3 & ^CHAN 1 GP 2^$PROCESS$ & $CP 1 GP^0^1$ARM 1 GP 2$
156             -> CAN 1 GP -> NCAN 1 GP ;
157           PHASE 3 & ^CHAN 1 GP 2^$PROCESS$ & $CP 1 GP^0^1$ARM 1 GP 3$
158             -> CAN 1 GP -> NCAN 1 GP ;
159   (9)     PHASE 3 & ^CHAN[\^#2 3 4&]GP\^$PROCESS$
160                -> CHAN\^<1>GP\^<2>$<1> -> CHAN\^<1>GP\^<2> ;
161           PHASE 3 & KILL[$#'[ '] '  ARM DARM CP P R S AO GP ^ .
162                           0 1 2 3 4 '#
163                          &$CP[\^\^#1 2 3 4 AO GP&]$ ->
164                              MODIFY\^<1>\^<2>KILL -> MODIFY 1 ;
165                               ] -> ;
166           PHASE 3 & ^CHAN[$#1 2.&]AO$PROCESS$ -> BUFF$<1>AO$<2> ;
167           PHASE 3 & $PROCESS CP 1 AO^0$ & ^CHAN 1 AO R$
168             -> CAN 1 AO -> NCAN 1 AO ;
169           PHASE 3 & $PROCESS CP 1 AO^1$ & ^CHAN 1 AO R$PROCESS$
170             & CHAN 1 AO R$
```

```
171           -> CHAN 1 AO R$<5>$<3> -> CHAN 1 AO R$<5> ;
172         PHASE 3 & ^CHAN[$AO\^#1 2 3 4 .
173                         & 1 AO\^ -> , AO . ;
174            ]$PROCESS$ -> CHAN$<1>AO\^<1>$<2> -> CHAN$<1>AO\^<1>;
175         PHASE 3 & ^START$PROCESS CP 1 PR$ -> PROCESS$<1> ;
176         PHASE 3 & ^CHAN[\^#2&]PR\^$PROCESS$
177             ->CHAN\^<1>PR\^<2>$<1> -> CHAN\^<1>PR\^<2> ;
178         PHASE 3 & ^MOVE\^CP[$#1 2 .&]AO$
179             -> BUFF 1 PR$<1>^0 MOVE\^<1>CP$<1>AO$<2> ;
180         PHASE 3 & ^MOVE\^$ -> PROCESS$<1> ;
181         PHASE 3 & ^[\^#MOVE NEW&]$ -> CAN 1 PR -> NCAN 1 PR ;
182         PHASE 3 & ^NEW$PROCESS CP[$#1 2.&]AO$ -> BUFF 1 PR$<2>
183             ^0 NEW 0$<1> ;
184         PHASE 3 &^CHAN$PROCESS CP[$#1 2 .&]AO$ -> BUFF 1 PR$<2>^ ;
185         PHASE 3 & PROCESS CP[$#1 2 .&]AO$ -> BUFF 1 PR$<1>^ ;
186       ]
```

SYSTEM C

```
1   (1)  C: PHASE 3 & CHAN 1.1 AO 1 & CHAN 1.1 AO R
2          & CHAN 1.2 AO 1 & CHAN 1.2 AO R
3          & CHAN 1.1 AO S & CHAN 1.2 AO S
4          & CHAN 2 PR S & CHAN 2 PR 1
5          & CHAN 2 GP 1 & BUFF 2 GP 1
6          & CHAN 2 GP 2 & BUFF 2 GP 2
7          & CHAN 2 GP 3 & BUFF 2 GP 3
8          & CHAN 3 GP 1 & BUFF 3 GP 1
9          & CHAN 3 GP 2 & BUFF 3 GP 2
10         & CHAN 4 GP 1 & BUFF 4 GP 1
11         & CHAN 4 GP 2 & BUFF 4 GP 2
12        & MODIFY 2
13         & PROCESS CP 2 PR^1 PR'[S1']
14         & PROCESS CP 1.1 AO^0 AO'[ CP 2 GP^0^1^1
15           DARM 2 GP 1 DARM 2 GP 2 DARM 2 GP 3 GP'[']']
16         & PROCESS CP 1.2 AO^0 AO'[CP 4 GP ^0^1^1 DARM
17          4 GP 1 DARM 4 GP 2 GP'[']] CP 3 GP ^0^1^1 DARM 3 GP 1
18         DARM 3 GP 2 GP'[']']
19  (2)  # PHASE 0 1 2 3 4 CHAN BUFF PROCESS ARM DARM CAN
20         NCAN  MODIFY KILL PR AO GP CP P R S ^ '[ '] '  ': ! .
21         MOVE START NEW TEST '# S1 S2
22  (3)  & PHASE 1 -> PHASE 2 ;
23         PHASE 2 -> PHASE 3 ;
24         PHASE 3 -> PHASE 1 ;
25         $ -> @ S2' ' $<1> -> PRINT 2 ;
26  (4)    CHAN\^GP\^$ -> CHAN\^<1>GP\^<2> ;
27         PHASE 3 & $PROCESS CP[$#1 2 .&]AO $DARM\^\^\^$
28             -> BUFF\^<1>\^<2>\^<3> ;
29         PHASE \3 & CHAN[$#1 2.&]C\GP#AO PR$]$ -> CHAN$<1>\GP<1>$<2> ;
30         PHASE 3 & $PROCESS CP 1.1 AO^1$ & CHAN 1.1 AO$
31           -> CHAN 1.1 AO$<3> ;
32         PHASE 3 & $PROCESS CP 1.2 AO^1$ & CHAN 1.2 AO$
33           -> CHAN 1.2 AO$<3> ;
34         PHASE 3 & ^MOVE\^CP 1.1 AO$ & CHAN 1.1 AO$
35           -> CHAN 1.1 AO $<2> ;
36         PHASE 3 & ^MOVE\^CP 1.2 AO$ & CHAN 1.2 AO$
37           -> CHAN 1.2 AO$<2> ;
38         PHASE 3 & CHAN\^PR\^^$ -> BUFF\^<1>PR\^<2>^\^<2>$<1> ;
39         PHASE 3 & CHAN\^PR\^ -> BUFF\^<1>PR\^<2>^ ;
40         PHASE 3 & CHAN \^ PR\^$ -> CHAN \^<1> PR\^<2> ;
41         PHASE 3 & $PROCESS CP 1.1 AO^0$ & CHAN 1.1 AO$
42           -> BUFF 1.1 AO$<3> ;
43         PHASE 3 & $PROCESS CP 1.2 AO^0$ & CHAN 1.2 AO$
44           -> BUFF 1.2 AO$<3> ;
45         PHASE 3 & $PROCESS CP 1.1 AO^0$ & CHAN 1.1 AO\^$
46           -> CHAN 1.1 AO\^<1> ;
47         PHASE 3 & $PROCESS CP 1.2 AO^0$ & CHAN 1.2 AO\^$
48           -> CHAN 1.2 AO\^<1> ;
49         CHAN 2 GP 1 & BUFF 2 GP 1$ & $PROCESS$ARM 2 GP 1$
50           -> BUFF 2 GP 1$<1> ;
51         CHAN 2 GP 2 & BUFF 2 GP 2$ & $PROCESS$ARM 2 GP 2$
52           -> BUFF 2 GP 2$<1> ;
53         CHAN 2 GP 3 & BUFF 2 GP 3$ & $PROCESS$ARM 2 GP 3$
54           -> BUFF 2 GP 3$<1> ;
55         CHAN 3 GP 1 & BUFF 3 GP 1$ & $PROCESS$ARM 3 GP 1$
56           -> BUFF 3 GP 1$<1> ;
```

```
57          CHAN 3 GP 2 & BUFF 3 GP 2$ & $PROCESS$ARM 3 GP 2$
58            -> BUFF 3 GP 2$<1> ;
59          CHAN 4 GP 1 & BUFF 4 GP 1$ & $PROCESS$ARM 4 GP 1$
60            -> BUFF 4 GP 1$<1> ;
61          CHAN 4 GP 2 & BUFF 4 GP 2$ & $PROCESS$ARM 4 GP 2$
62            -> BUFF 4 GP 2$<1> ;
63          CHAN 2 GP 1^$ & $PROCESS$ARM 2 GP 1$ -> BUFF 2 GP 1^$<1> ;
64          CHAN 2 GP 2^$ & $PROCESS$ARM 2 GP 2$ -> BUFF 2 GP 2^$<1> ;
65          CHAN 2 GP 3^$ & $PROCESS$ARM 2 GP 3$ -> BUFF 2 GP 3^$<1> ;
66          CHAN 3 GP 1^$ & $PROCESS$ARM 3 GP 1$ -> BUFF 3 GP 1^$<1> ;
67          CHAN 3 GP 2^$ & $PROCESS$ARM 3 GP 2$ -> BUFF 3 GP 2^$<1> ;
68          CHAN 4 GP 1^$ & $PROCESS$ARM 4 GP 1$ -> BUFF 4 GP 1^$<1> ;
69          CHAN 4 GP 2^$ & $PROCESS$ARM 4 GP 2$ -> BUFF 4 GP 2^$<1> ;
70    (5)   PHASE 1 & PROCESS CP[$#1 2 .&]AO^1$
71              -> PROCESS CP$<1>':AO^1$<2> ;
72          CAN 2 PR & PROCESS CP 2 PR^\^$  & BUFF 2 PR 1^$
73            & BUFF 2 PR S^$
74            & BUFF 2 PR 1.1^$ & BUFF 2 PR 1.2^$
75            -> !$<2>!$<3>!$<4>!$<5>!PROCESS CP 2':PR^1$<1> ;
76          NCAN 2 PR & PROCESS CP 2 PR^1$ -> !PROCESS CP 2 ':PR^1$<1> ;
77          NCAN 2 PR & PROCESS CP 2 PR^0$ -> ':PROCESS CP 2 PR^0$<1> ;
78           CAN 1.1 AO & PROCESS CP 1.1 AO^0$ & BUFF 1.1 AO 1$
79             & BUFF 1.1 AO S
80               & BUFF 1.1 AO R$ -> $<2>$<3>$<4>PROCESS CP 1.1':AO^1$<1> ;
81           CAN 1.2 AO & PROCESS CP 1.2 AO^0$ & BUFF 1.2 AO 1$
82             & BUFF 1.2 AO S$
83               & BUFF 1.2 AO R$ -> $<2>$<3>$<4>PROCESS CP 1.2':AO^1$<1> ;
84           NCAN 1.1 AO & CAN 2 GP & $CP 2 GP^1$
85                 -> $<1>CP 2':GP^1$<2> ;
86           NCAN 1.1 AO & CAN 2 GP & $CP 2 GP^0^1^0$
87                 & BUFF 2 GP 1^$ -> ^$<3>$<1>CP 2':GP^1^1^0$<2> ;
88           NCAN 1.1 AO & CAN 2 GP & $CP 2 GP^0^1^0$ & CHAN 2 GP 1$
89               & BUFF 2 GP 1^$ -> BUFF 2 GP 1$<3> -> BUFF 2 GP 1^$<4> ;
90           NCAN 1.1 AO & CAN 2 GP & $CP 2 GP^0^1^0$
91             & BUFF 2 GP 1 & BUFF 2 GP 2^$
92             -> ^$<3>$<1>CP 2':GP^1^1^0$<2> ;
93           NCAN 1.1 AO & CAN 2 GP & $CP 2 GP^0^1^0$
94             & BUFF 2 GP 1 & CHAN 2 GP 2$ & BUFF 2 GP 2^$
95             -> BUFF 2 GP 2$<3> -> BUFF 2 GP 2^$<4> ;
96           NCAN 1.1 AO & CAN 2 GP & $CP 2 GP^0^1^0$
97             & BUFF 2 GP 1 & BUFF 2 GP 2 & BUFF 2 GP 3^$
98             ->^$<3>$<1>CP 2 ':GP^1^1^0$<2> ;
99           NCAN 1.1 AO & CAN 2 GP & $CP 2 GP^0^1^0$
100            & BUFF 2 GP 1 & BUFF 2 GP 2
101            & CHAN 2 GP 3$ & BUFF 2 GP 3^$
102            -> BUFF 2 GP 3$<3> -> BUFF 2 GP 3^$<4> ;
103          NCAN 1.1 AO & CAN 2 GP & $CP 2 GP^0^1^1$
104            & BUFF 2 GP 1$ & BUFF 2 GP 2$ & BUFF 2 GP 3$
105              -> $<3>$<4>$<5>$<1>CP 2':GP^1^1^1$<2> ;
106          NCAN 1.1 AO & CAN 2 GP & $CP 2 GP^0^1^1$ & CHAN 2 GP 1$
107               & BUFF 2 GP 1$
108               -> BUFF 2 GP 1$<3> -> BUFF 2 GP 1$<4> ;
109          NCAN 1.1 AO & CAN 2 GP & $CP 2 GP^0^1^1$
110            &CHAN 2 GP 2$ & BUFF 2 GP 2$
111            -> BUFF 2 GP 2$<3> -> BUFF 2 GP 2$<4> ;
112          NCAN 1.1 AO & CAN 2 GP & $CP 2 GP^0^1^1$
113            & CHAN 2 GP 3$ & BUFF 2 GP 3$
```

```
114             -> BUFF 2 GP 3$<3> -> BUFF 2 GP 3$<4> ;
115           NCAN 1.1 AO & NCAN 2 GP & PROCESS CP 1.1 AO$
116             -> ':PROCESS CP 1.1 AO$<1> ;
117           NCAN 1.2 AO & CAN 3 GP & $CP 3 GP^1$
118             -> $<1>CP 3 ':GP^1$<2> ;
119           NCAN 1.2 AO & CAN 3 GP & $CP 3 GP^0^1^0$
120             & BUFF 3 GP 1^$
121             -> ^$<3>$<1>CP 3':GP^1^1^0$<2> ;
122           NCAN 1.2 AO & CAN 3 GP & $CP 3 GP^0^1^0$
123             & CHAN 3 GP 1$ & BUFF 3 GP 1^$
124             -> BUFF 3 GP 1$<3> -> BUFF 3 GP 1^$<4> ;
125           NCAN 1.2 AO & CAN 3 GP & $CP 3 GP^0^1^0$
126             & BUFF 3 GP 1 & BUFF 3 GP 2^$
127             -> ^$<3>$<1>CP 3':GP^1^1^0$<2> ;
128           NCAN 1.2 AO & CAN 3 GP & $CP 3 GP^0^1^0$
129             & BUFF 3 GP 1 & CHAN 3 GP 2 $ & BUFF 3 GP 2^$
130             -> BUFF 3 GP 2$<3> -> BUFF 3 GP 2^$<4> ;
131           NCAN 1.2 AO & CAN 3 GP & $CP 3 GP^0^1^1$
132             & BUFF 3 GP 1$ & BUFF 3 GP 2$
133             -> $<3>$<4>$<1>CP 3 ':GP^1^1^1$<2> ;
134           NCAN 1.2 AO & CAN 3 GP & $CP 3 GP^0^1^1$
135             & CHAN 3 GP 1$ & BUFF 3 GP 1$
136             -> BUFF 3 GP 1$<3> -> BUFF 3 GP 1$<4> ;
137           NCAN 1.2 AO & CAN 3 GP & $CP 3 GP^0^1^1$
138             & CHAN 3 GP 2$ & BUFF 3 GP 2$
139             -> BUFF 3 GP 2$<3> -> BUFF 3 GP 2$<4> ;
140           NCAN 1.2 AO & NCAN 3 GP & CAN 4 GP
141             & $CP 4 GP^1$
142             -> $<1>CP 4':GP^1$<2>;
143           NCAN 1.2 AO & NCAN 3 GP & CAN 4 GP
144             & $CP 4 GP^0^1^0$ & BUFF 4 GP 1^$
145             ->^$<3>$<1>CP 4':GP^1^1^0$<2> ;
146           NCAN 1.2 AO & NCAN 3 GP & CAN 4 GP
147             & $CP 4 GP^0^1^0$ & CHAN 4 GP 1$ & BUFF 4 GP 1^$
148             -> BUFF 4 GP 1$<3> -> BUFF 4 GP 1^$<4> ;
149           NCAN 1.2 AO & NCAN 3 GP & CAN 4 GP
150             & $CP 4 GP ^0^1^0$ & BUFF 4 GP 1 & BUFF 4 GP 2^$
151             ->^$<3>$<1>CP 4':GP^1^1^0$<2> ;
152           NCAN 1.2 AO & NCAN 3 GP & CAN 4 GP
153             & $CP 4 GP^0^1^0$
154             & BUFF 4 GP 1 & CHAN 4 GP 2$ & BUFF 4 GP 2^$
155             -> BUFF 4 GP 2$<3> -> BUFF 4 GP 2^$<4> ;
156           NCAN 1.2 AO & NCAN 3 GP & CAN 4 GP
157             & $CP 4 GP^0^1^1$ & BUFF 4 GP 1$ & BUFF 4 GP 2$
158             -> $<3>$<4>$<1>CP 4':GP^1^1^1$<2> ;
159           NCAN 1.2 AO & NCAN 3 GP & CAN 4 GP
160             & $CP 4 GP^0^1^1$
161             & CHAN 4 GP 1$ & BUFF 4 GP 1$
162             -> BUFF 4 GP 1 $<3> -> BUFF 4 GP 1$<4> ;
163           NCAN 1.2 AO & NCAN 3 GP & CAN 4 GP
164             & $CP 4 GP^0^1^1$
165             & CHAN 4 GP 2$ & BUFF 4 GP 2$
166             -> BUFF 4 GP 2 $<3> -> BUFF 4 GP 2$<4> ;
167           NCAN 1.2 AO & NCAN 3 GP & NCAN 4 GP
168             & PROCESS CP 1.2 AO$
169             -> ':PROCESS CP 1.2 AO $<1> ;
170  (6)      PHASE 2 & ':PROCESS$ -> PROCESS$<1> ;
```

```
171          PHASE 3 & $PROCESS$ -> PROCESS$<2> ;
172          PHASE 2 -> MOOIFY 2;
173  (7)     PHASE 2 & [$';PR$#PROCESS ARM DARM CP P R S PR AO GP ^ O
174                    1 2 3 4 '[ '] '  START MOVE NEW TEST ! . CHAN
175                    '# '; S1 S2
176            &!\^NEW\^CP[$#1 2.&]AO$CP[\^#1 2 3 4&]GP$';$
177                      -> MOOIFY$<1>NEW\^<3>GP -> MOOIFY 2 ;
178             !PROCESS$PR^1 PR'[S1$ -> PROCESS$<1>PR^O PR'[S1$<2> ;
179             ![\^$#ARM OARM CP P R S AO GP ^ O 1 2 3 4 '[ '] . '# S1 S2
180                '  TEST MOVE NEW&]!$';$S1$ -> !$<2>';$<3>\^<1>$<1>!S1$<4> ;
181             !!$';$ -> !$<1>';$<2> ;
182             !PROCESS$PR'[O NEW O[$#ARM DARM CP P R S AO GP CP PR ^ .
183                    O 1 2 3 4 '[ '] '  TEST MOVE NEW '# S1 S2
184                &]!$ -> ^START$<2>PROCESS$<1>PR'[$<3> ;
185             !PROCESS$PR'[\O NEW\O[$#ARM OARM CP P R S
186                  AO GP ^ O 1 2 3 4 '[ '] '  .&]!$
187              -> ^CHAN\O<1>PR S^R PROCESS$<1>
188                    PR'[START$<2>!$<3> ;
189             !PROCESS$PR'[START[$#ARM OARM CP P R S AO GP ^
190                  O 1 2 3 4 '[ '] '  . S1 S2&]!$
191              ->^START$<2>PROCESS$<1>PR'[$<3> ;
192             !PROCESS$PR'[\^'#!$
193              -> ^CHAN\^<1>PR S^S PROCESS$<1>PR'[$<2>;
194             !PROCESS$PR'[O MOVE\O[$#ARM DARM CP P R S
195                 AO GP ^ O 1 2 3 4 '[ '] '  . &]!$S1\^$']
196              ->PROCESS$<1>PR'[$<3>O MOVE\O<1>$<2>!S1\^<1>$<4>'] ;
197             !PROCESS$PR'[O MOVE\O[PC$#1 2 .&]AO[$
198                  # ARM DARM CP P R S AO GP ^ O 1 2 3 4
199                    '[ '] '  . &]!$S1']
200                ->^CHAN\O<1>PR 2^NEW 2 CP$<2>AO$<3>
201                  PROCESS$<1>PR'[$<4>S1$<2>'] ;
202             !PROCESS$PR'[S[\^#R S&]!$S1$']
203               ->^CHAN$<3>AO R^\^<1>PROCESS$<1>PR'[$<2>S1'] ;
204                  ] -> $<1>PR$<2> ;
205          PHASE 2 & [$';AO$#PROCESS ARM DARM P R S CP AO GP ^ O 1
206                  2 3 4 '[ '] '  ': '# . TEST
207                &^$PROCESS CP[$#1 2 .&]';AO^1$ -> PROCESS CP$<2>';AO^1$<3> ;
208                PROCESS CP[$#1 2 .&]';AO^1$ -> PROCESS CP$<1>';AO^O$<2> ;
209                  ] -> $<1>AO$<2> ;
210          PHASE 2 & [$';GP$ #PROCESS ARM DARM P R S AO GP ^ O 1
211                    TEST CP 2 3 4 '[ '] '  '# .
212                    & ^$PROCESS$';GP^1$ -> PROCESS$<2>';GP^1$<3> ;
213                    PROCESS$';GP^1$ -> PROCESS$<1>';GP^O$<2> ;
214                  ] -> $<1>GP$<2> ;
215  (8)     PHASE 3 & $PROCESS CP[$#1 2 .&]AO^1$ -> CAN $<2>AO ;
216          PHASE 3 & $PROCESS CP[$#1 2 .&]AO^O$ -> NCAN $<2> AO ;
217          PHASE 3 & $PROCESS CP[$#1 2 .&]AO$CP[\^#1 2 3 4&]GP^1$
218              ->CAN \^<1>GP ;
219          PHASE 3 & $PROCESS CP[$#1 2 .&]AO$CP[\^#1 2 3 4&]GP^O$
220              -> NCAN \^<1> GP ;
221          PHASE 3 & ^MOVE$CP[$#1 2 3 4 .&]E\^#AO GP&]$
222            -> NCAN$<2>\^<1> ;
223          PHASE 3 -> NCAN 2 PR ;
224          PHASE 3 & CHAN[$#1 2 .&]E\GP#AO PR&]\^^$
225              -> CAN$<1>\GP<1> -> NCAN$<1>\GP<1> ;
226          PHASE 3 & E\^#CHAN BUFF&] 2 GP 1^$ & $CP 2 GP^O^1$ARM 2 GP 1$
227                -> CAN 2 GP -> NCAN 2 GP ;
```

```
228         PHASE 3 & [\^#CHAN BUFF$]2 GP 2^$ & $CP 2 GP^0^1$ARM 2 GP 2$
229           -> CAN 2 GP -> NCAN 2 GP ;
230         PHASE 3 & [\^#CHAN BUFF$]2 GP 3^$ & $CP 2 GP^0^1$ARM 2 GP 3$
231           -> CAN 2 GP -> NCAN 2 GP ;
232         PHASE 3 & [\^#CHAN BUFF$]3 GP 1^$ & $CP 3 GP^0^1$ARM 3 GP 1$
233           -> CAN 3 GP -> NCAN 3 GP ;
234         PHASE 3 & [\^#CHAN BUFF$]3 GP 2^$ & $CP 3 GP^0^1$ARM 3 GP 2$
235           -> CAN 3 GP -> NCAN 3 GP ;
236         PHASE 3 & [\^#CHAN BUFF$]4 GP 1^$ & $CP 4 GP^0^1$ARM 4 GP 1$
237           -> CAN 4 GP -> NCAN 4 GP ;
238         PHASE 3 & [\^#CHAN BUFF$]4 GP 2^$ & $CP 4 GP^0^1$ARM 4 GP 2$
239           -> CAN 4 GP -> NCAN 4 GP ;
240         PHASE 3 & ^CHAN 2 GP 1^$PROCESS$ & $CP 2 GP^0^1$ARM 2 GP 1$
241            & [\^#CHAN BUFF$]2 GP 1$
242            -> BUFF 2 GP 1^$<1> -> BUFF 2 GP 1$<6> ;
243         PHASE 3 & ^CHAN 2 GP 2^$PROCESS$ & $CP 2 GP^0^1$ARM 2 GP 2$
244           & [\^#CHAN BUFF$]2 GP 2$
245           -> BUFF 2 GP 2^$<1> -> BUFF 2 GP 2$<6> ;
246         PHASE 3 & ^CHAN 2 GP 3^$PROCESS$
247           & $CP 2 GP^0^1$ARM 2 GP 3$ & [\^#CHAN BUFF$]2 GP 3$
248           -> BUFF 2 GP 3^$<1> -> BUFF 2 GP 3$<6> ;
249         PHASE 3 & ^CHAN 3 GP 1^PROCESS$
250           & $CP 3 GP^0^1$ARM 3 GP 1$ & [\^#CHAN BUFF$]3 GP 1$
251           -> BUFF 3 GP 1^$<1> -> BUFF 3 GP 1$<6> ;
252         PHASE 3 & ^CHAN 3 GP 2^PROCESS$
253           & $CP 3 GP^0^1$ARM 3 GP 2$ & [\^#CHAN BUFF$]3 GP 2$
254           -> BUFF 3 GP 2^$<1> -> BUFF 3 GP 2$<6> ;
255         PHASE 3 & ^CHAN 4 GP 1^$PROCESS$
256           & $CP 4 GP^0^1$ARM 4 GP 1$ & [\^#CHAN BUFF$]4 GP 1$
257           -> BUFF 4 GP 1^$<1> -> BUFF 4 GP 1$<6> ;
258         PHASE 3 & ^CHAN 4 GP 2^$PROCESS$
259           & $CP 4 GP^0^!$ARM 4 GP 2$ & [\^#CHAN BUFF$]4 GP 2$
260           -> BUFF 4 GP 2^$<1> -> BUFF 4 GP 2$<6> ;
261         PHASE 3 & ^CHAN 2 GP 1^$PROCESS$ & $CP 2 GP^0^1$ARM 2 GP 1$
262               -> CAN 2 GP -> NCAN 2 GP ;
263         PHASE 3 & ^CHAN 2 GP 2^$PROCESS$ & $CP 2 GP^0^1$ARM 2 GP 2$
264           -> CAN 2 GP -> NCAN 2 GP ;
265         PHASE 3 & ^CHAN 2 GP 3^$PROCESS$ & $CP 2 GP^0^1$ARM 2 GP 3$
266           -> CAN 2 GP -> NCAN 2 GP ;
267         PHASE 3 & ^CHAN 3 GP 1^$PROCESS$ & $CP 3 GP^0^1$ARM 3 GP 1$
268           -> CAN 3 GP -> NCAN 3 GP ;
269         PHASE 3 & ^CHAN 3 GP 2^$PROCESS$ & $CP 3 GP^0^1$ARM 3 GP 2$
270           -> CAN 3 GP -> NCAN 3 GP ;
271         PHASE 3 & ^CHAN 4 GP 1^$PROCESS$ & $CP 4 GP^0^1$ARM 4 GP 1$
272           -> CAN 4 GP -> NCAN 4 GP ;
273         PHASE 3 & ^CHAN 4 GP 2^$PROCESS$ & $CP 4 GP^0^1$ARM 4 GP 2$
274           -> CAN 4 GP -> NCAN 4 GP ;
275   (9)   PHASE 3 & ^CHAN[\^#1$]GP\^$PROCESS$
276               -> CHAN\^<1>GP\^<2>$<1> -> CHAN\^<1>GP\^<2> ;
277         PHASE 3 & KILL[$#'[ ']  ' ARM DARM CP P R S AO GP ^ .
278                     0 1 2 3 4 '#
279                         $$CP[\^\^#1 2 3 4 AO GP$]$ ->
280                             MODIFY\^<1>\^<2>KILL -> MODIFY 2 ;
281                              ] -> ;
282         PHASE 3 & ^CHAN[$#1 2.$]AO$PROCESS$ -> BUFF$<;>AO$<2> ;
283         PHASE 3 & $PROCESS CP 1.1 AO^O$ & ^CHAN 1.1 AO$
284           -> CAN 1.1 AO -> NCAN 1.1 AO ;
```

```
285         PHASE 3 & $PROCESS CP 1.2 AO^0$ & ^CHAN 1.2 AO$
286           -> CAN 1.2 AO -> NCAN 1.2 AO ;
287         PHASE 3 & $PROCESS CP 1.1 AO^1$ & ^CHAN 1.1 AO R$PROCESS$
288           & CHAN 1.1 AO R$
289           -> CHAN 1.1 AO R$<5>$<3> -> CHAN 1.1 AO R$<5> ;
290         PHASE 3 & $PROCESS CP 1.2 AO^1$ & ^CHAN 1.2 AO R$PROCESS$
291           & CHAN 1.2 AO R$
292           -> CHAN 1.2 AO R$<5>$<3> -> CHAN 1.2 AO R$<3> ;
293         PHASE 3 & ^CHAN[$AO\^#1 2 3 4 .
294                         & 1.1 AO\^ -> . AO . ;
295                           1.2 AO\^ -> . AO . ;
296            ]$PROCESS$ -> CHAN$<1>AO\^<1>$<2> -> CHAN$<1>AO\^<1>;
297         PHASE 3 & ^START$PROCESS CP 2 PR$ -> PROCESS$<1> ;
298         PHASE 3 & ^CHAN[\^#1&]PR\^$PROCESS$
299             ->CHAN\^<1>PR\^<2>$<1> -> CHAN\^<1>PR\^<2> ;
300         PHASE 3 & ^MOVE\^CP[$#1 2 .&]AO$
301             -> BUFF 2 PR$<1>^0 MOVE\^<1>CP$<1>AO$<2> ;
302         PHASE 3 & ^MOVE\^$ -> PROCESS$<1> ;
303         PHASE 3 & ^[\^#MOVE NEW&]$ -> CAN 2 PR -> NCAN 2 PR ;
304         PHASE 3 & ^NEW$PROCESS CP[$#1 2 .&]AO$ -> BUFF 2 PR$<2>
305                 ^0 NEW 0$<1> ;
306         PHASE 3 & ^CHAN$PROCESS CP[$#1 2 .&]AO$ -> BUFF 2 PR$<2> ^ ;
307         PHASE 3 & PROCESS CP[$#1 2 .&]AO$ -> BUFF 2 PR$<1>^ ;
308       ]
```

BIBLIOGRAPHY

Al 71 Alsberg, P. A. OSL/2 - An Operating System Language. Ph.D. Thesis. Champain, Illinois: University of Illinois at Urbana, 1971.

BN 71 Bell, C. Gorden, and Newell, Allen. Computer Structures: Readings and Examples. New York: McGraw-Hill Book Company, 1971.

BC 69 Bensoussan, A., C. T. Clingen and R. C. Daley. "The Multics Virtual Memory," Association for Computing Machinery. Second Symposium on Operating System Principles. Princeton University, 1969. 30-42.

BD 69 Bernstein, A. J., G. D. Detlefsen and R. H. Kerr. "Process Control and Communication," Association for Computing Machinery. Second Symposium on Operating System Principles. Princeton University, 1969. 60-6.

Be 71 Berry, Daniel M. "Tasking in Oregano," Proceedings of the Fifth Annual Princeton Conference on Information Sciences and Systems. Department of Electrical Engineering, Princeton University, 1971. 295-9.

BB 69 Bétourné, C.; J. Boulenger, J. Ferrié, C. Kaiser, J. Kott, S. Krakowiak and J. Mossiére. "Process Management and Resource Sharing in the Multiaccess System 'ESOPE,'" Association for Computing Machienry. Second Symposium on Operating Systems Principles. Princeton University, 1969. 67-74.

BB 71 Bobrow, D. G., J. D. Burchfiel, D. L. Murphy and R. S. Tomlinson. "TENEX, A Paged Time Sharing System for the PDP-10," Association for Computing Machinery. Third Symposium on Operating Systems Principles. Stanford University, 1971. 1-10.

Br 72 Bradshaw, F. T. "Some Structural Ideas for Computer Systems," IEEE Computer Society. COMPCON 72 - Digest of Papers: Innovative Architecture. San Francisco, Cal., 1972. 183-186.

CC 70 Carr, C. Stephen, Stephen D. Crocker and Vinton G. Cerf. "HOST-HOST Communication Protocol in the ARPA network," AFIPS Conference Proceedings. Volume 36, 1970 Spring Joint Computer Conference. Montvale, New Jersey: AFIPS Press, 1970. 589-97.

CE 71 Coffman, E. G., Jr., M. J. Elphick and A. Shoshani. "System Deadlocks," Computing Surveys, 3, 2 (June 1971), 67-68.

Co 71 Commission on Education of the National Academy of Engineering. Interim report of the COSINE Committee. An Undergraduate Course on Operating Systems Principles. Washington, D.C. 1971.

CM 72 Conway, R. W., W. L. Maxwell and H. L. Morgan. "On the Implementation of Security Measures in Information Systems," Communications of the ACM, 15, 4 (April 1972). 211-20.

Ca 73 Cowan, George Jr. Management of Resources in a Potentially Hostile Environment (Logical and Physical). Ph.D. Thesis in preparation, Computer Sciences Department, University of Wisconsin.

CH 72 Crocker, Steven D., John F. Heafner, Robert M. Metcalfe and Jonathan B. Postel. "Function-Oriented Protocols for the ARPA Computer Network," AFIPS Conference Proceedings, Vol. 36, 1972 Spring Joint Computer Conference (Montvale, New Jersey: AFIPS Press, 1972). 271-279.

DM 68 Dahl, Ole-Johan, Bjorn Myrhaug and Kristen Nygaard. Simula 67 - Common Base Language. Oslo, Norway: Norwegian Computing Center, 1968.

DN 67 Dahl, Ole-Johan and Nygaard, Kristen. Simula - A Language for Programming and Description of Discrete Event Systems. Oslo 3, Norway: Norwegian Computing Center, 1967.

Da 72 Datamation. Communications, "ARPA Network to go Commercial." 18, 4 (April 1972), 106.

De 68 Denning, Peter James. Resource Allocation in Multiprocess Computer Systems. Ph.D. Thesis. Massachusetts Institute of Technology, 1968.

De 71 Denning, Peter J. "Third Generation Computer Systems," Computing Surveys. 3, 4 (December 1971), 175-216.

De 70 Dennis, Jack B. "Forward." Association for Computing Machinery. Record of the Project MAC Conference on Concurrent Systems and Parallel Computation. Woods Hole, Mass., 1970. v-vii.

DV 66 Dennis, Jack B. and Van Horn, Earl C. "Programming Semantics for Multiprogrammed Computations," Communications of the ACM, 9, 3 (March, 1966), 143-55.

Di 65 Dijkstra, E. W. "Solution of a Problem in Concurrent Programming Control," Communications of the ACM. 8, 9 (September 1965), 569.

Di 67 Dijkstra, E. W. "Cooperating Sequential Processes," Programming Languages. Edited by F. Genuys. New York: Academic Press, 1967.

Di 68 Dijkstra, Edsger W. "The Structure of the 'THE'--Multiprogramming System," Communications of the ACM. 11, 5 (May 1968), 341-6.

Di 71 Dijkstra, E. W. "Hierarchial Ordering of Sequential Processes, ACTA Informatica. 1, 2 (1971) New York: Springer-Verlag, 1971. 115-38.

Er 71 Ershov, Andrei P. Parallel Programming. Computer Science Department Report No. CS-224, Stanford, Cal.: Computer Science Department, Stanford University, 1971.

Fa 72 Farber, David J. "Networks: An Introduction," Datamation, 18, 4 (April 1972), 36-9.

FO 71 Feiertag, R. J. and Organick, E. I. "The Multics Input/Output System," Association for Computing Machinery. Third Symposium on Operating Systems Principles. Stanford University, 1971. 35-41.

Fi 70 Fisher, David A. Control Structures for Programming Languages. Ph.D. Thesis. Computer Science Department. Pittsburgh, Pa.: Carnegie-Mellon University, 1970.

FH 71 Fitzwater, D. R. and Hintz, C. A. A System for the Formal Definition of Digital Systems. Computer Sciences Technical Report #141. Madison, Wis.: The University of Wisconsin Computer Sciences Department, 1971.

Fi 73 Fitzwater, D. R. "Design and Formal Definition of an Abstract High Level Digital System." Working document. Computer Sciences Department, University of Wisconsin, Madison, Wisconsin.

FM 65a Fitzwater, D. R. and McFarland, D.E. Task 65 - A Consequent Procedure Real Time Computer Language. United States Atomic Energy Commission Research and Development Report IS-1280. Springfield, Virginia: Clearinghouse for Federal Scientific and Technical Information, 1965.

FM 65b Fitzwater, D.R., D.E. McFarland and C.E. Runge. SDS 910 Implementation of the Task 65 Real Time Programming Language. United States Atomic Energy Commission Research and Development Report IS-1279. Springfield, Virginia: Clearinghouse for Federal Scientific and Technical Information, 1965.

Fo 71 Fontao, Rafael O. "A Concurrent Algorithm for Avoiding Deadlocks in Multiprocess Multiple Resource Systems," Association for Computing Machinery. Third Symposium on Operating Systems Principles. Stanford University, 1971. 72-9.

Ga 71 Gaines, R. Stockton. "An Operating System Based on the Concept of a Supervisory Computer." Association for Computing Machinery. Third Symposium on Operating Systems Principles. Stanford University, 1971. 17-23.

Gl 72a Glaser, E. L. "Introduction and Overview of the LOGOS Project." IEEE Computer Society. COMPCON 72 - Digest of Papers: Innovative Architecture. San Francisco, Cal., 1972. 175-7.

Gl 72b Glaser, E. L. "LOGOS--Where it is now and where it is going." IEEE Computer Society. COMPCON 72 - Digest of Papers: Innovative Architecture. San Francisco, Cal. 1972. 191-2.

Gr 71a Graham, Gordon Scott. Protection Structures in Operating Systems. Master of Science Thesis. Toronto, Ont., Canada: University of Toronto, 1971.

Gr 68 Graham, Robert M. "Protection in an Information Processing Utility," Communications of the ACM, 11, 5 (May 1968), 365-9.

Gr 71b Grant, Charles Alexander. Command Communication between Processes. Ph.D. Thesis. Berkeley, Calif.: University of California, Berkeley, 1971.

Ha 70 Hansen, Per Brinch. "The Nucleus of a Multiprogramming System," Communication of the ACM, 13, 4 (April 1970), 238-41.

Ha 71a Hansen, Per Brinch, ed. RC 4000 Software Multiprogramming System. Copenhaugen: Als Regnecentralen, 1971.

Ha 71b Hansen, Per Brinch. "Short-Term Scheduling in Multiprogramming Systems," Association for Computing Machinery. Third Symposium on Operating Systems Principles. Stanford University, 1971. 101-5.

Ha 68 Havender, J. W. "Avoiding Deadlock in Multi-Tasking Systems," IBM Systems Journal 7, 2 (1968), 74-84.

HK 70 Heart, F. E., R. E. Kahn, S. M. Ornstein, W. R. Crowder and D. C. Walder. "The Interface Message Processor for the ARPA Network," AFIPS Conference Proceedings. Volume 36, 1970 Spring Joint Computer Conference. Montvale, New Jersey: AFIPS Press, 1970. 551-67.

HR 72 Heath, F. G. and Rose, C. W. "The Case for Integrated Hardware/Software Design, with CAD Implications," IEEE Computer Society. COMPCON 72 - Digest of Papers: Innovative Architecture. San Francisco, Cal., 1972. 179-82.

He 70 Hebalkar, Prakash G. Deadlock - Free Sharing of Resources in Asynchronous Systems. Sc.D. Thesis. Project MAC Report MAC TR-75 (Thesis). Cambridge, Mass.: Project MAC, Massachusetts Institute of Technology, 1970.

He 71 Herriot, Robert George. The Definition of the Control and Environment Structure of Programming Languages. Ph.D. Thesis. Madison, Wisconsin: University of Wisconsin, 1971.

Ho 72 Hootman, Joseph T. "The Computer Network as a Market Place," Datamation, 18, 4 (April 1972), 43-6.

HR 69 Horning, J. J. and Randell, B. Structuring Complex Processes. IBM Research Report RC 2459. York Town Heights, New York: IBM Thomas J. Watson Research Center, 1969.

HR 71 Horning, J. J. and Randell, B. Process Structuring. Computer Systems Research Group. Toronto, Ont. Canada: University of Toronto, 1971.

IB 68 IBM "PL/I Language Specifications," IBM System Reference Library, Form No. C28-6571-4, IBM Corp. 1968.

Jo 73 Johnson, Robert T. Proving Assertions about State Structures of Formally Defined, Interacting, Digital Systems. Ph.D. Thesis to be available July 1973. Computer Sciences Department. University of Wisconsin.

La 69 Lampson, Butler W. "Dynamic Protection Structures," AFIPS Conference Proceedings. Volume 35 - 1969 Fall Joint Computer Conference. Montvale, New Jersey: AFIPS Press, 1969, 27-38.

La 71 Lampson, Butler W. "Protection," Proceedings of the Fifth Annual Princeton Conference on Information Sciences and Systems. Department of Electrical Engineering, Princeton University, 1971. 437-43.

La 66 Landin, P. J. "The Next 700 Programming Languages," Communications of the ACM. 9, 3 (March 1966), 157-66.

Li 71 Liskov, Barbara H. "The Design of the Venus Operating System," Association for Computing Machinery. Third Symposium on Operating Systems Principles. Stanford University, 1971. 11-16.

Lo 72 Lorin, Harold. Parallelism in Hardware and Software: Real and Apparent Concurrency. Englewood Cliffs, N.J.: Prentice-Hall Inc., 1972.

Ne 71 Needham, R. M. "Handling Difficult Faults in Operating Systems," Association for Computing Machinery. Third Symposium on Operating Systems Principles. Stanford University, 1971. 55-7.

Or 72 Organick, Elliott I. The Multics System: An Examination of its Structure. Cambridge, Mass.: The MIT Press, 1972.

OH 72 Ornstein, S.M., F.E. Heart, W.R. Crowther, H.K. Rising, S.B. Russell, and A. Michel. "The Terminal IMP for the ARPA Computer Network." AFIPS Conference Proceedings. Volume 40, 1972 Spring Joint Computer Conference. Montvale, New Jersey: AFIPS Press, 1972. 243-54.

Pa 73 Papazian, Vatche P. "Working Notes on Dynamic Processor Modifications," Computer Sciences Department, University of Wisconsin.

RW 70 Roberts, Lawrence G. and Wessler, Barry D. "Computer Network Development to Achieve Resource Sharing," AFIPS Conference Proceedings Volume 36, 1970 Spring Joint Computer Conference. Montvale, New Jersey: AFIPS Press, 1970. 543-9.

Ro 71 Rose, Charles William. A System of Representation for General Purpose Digital Computer Systems. Ph.D. Thesis. Cleveland, Ohio: Case Western Reserve University, 1971.

RB 72 Rose, C. W., F. T. Bradshaw and S. W. Katzke. "The LOGOS Representation System," IEEE Computer Society. COMPCON 72 - Digest of Papers: Innovative Architecture. San Francisco, Cal., 1972. 187-90.

Sc 65 Scherr, Allan Lee. An Analysis of Time-Shared Computer Systems. Ph.D. Thesis. Cambridge, Mass.: Massachusetts Institute of Technology, 1965.

SS 71 Schroeder, Michael D. and Saltzer, Jerome H. "A Hardware Architecture for Implementing Protection Rings," Association for Computing Machinery. Third Symposium on Operating Systems Principles. Stanford University, 1971. 42-54.

Se 70 Séror, Denis David. D.C.P.L. - A Distributed Control Programming Language. Ph.D. Thesis. Salt Lake City, Utah: University of Utah, 1970.

Sh 69 Shoshani, Arie. Detection, Prevention and Recovery from Deadlocks in Multiprocess Multiple Resource Systems. Princeton, N.J.: Princeton University, 1969.

SO 69 Spier, Michael J. and Organick, Elliott I. "The Multics Interprocess Communication Facility," Association for Computing Machinery. Second Symposium on Operating Systems Principles. Princeton University, 1969. 83-91.

St 72 Stefferud, Einar. "Management's Role in Networking," Datamation. 18, 4 (April 1972), 40-2.

TH 72 Thomas, Robert H. and Henderson, D. Austen. "McROSS - A Multi-Computer Programming System," AFIPS Conference Proceedings. Volume 40, 1972 Spring Joint Computer Conference. Montvale, New Jersey: AFIPS Press, 1972. 281-93.

Va 69 Vanderbilt, Dean Hanawalt. Controlled Information Sharing in a Computer Utility. Ph.D. Thesis. Cambridge, Mass.: Massachusetts Institute of Technology, 1969.

Va 66 VanHorn, Earl C., Jr. Computer Design for a Synchronously Reproducible Multiprocessing. Ph.D. Thesis. Massachusetts Institute of Technology, 1966.

Wa 70 Waite, W. M. "The Mobile Programming System: STAGE2," Communications of the ACM. 13, 7 (July 1970), 415-21.

Wi 69 Wirth, Niklous. "On Multiprogramming Machine Coding and Computer Organization," Communications of the ACM. 12, 9 (September 1969), 489-98.

Wi 72 Withington, Frederick G. "The Next (and Last?) Generation," Datamation. 18, 5 (May 1972), 71-4.

14 AUG 73
22 JUL 76
10 DEC 76
S10173
23499
23748

143377
Thesis
K848 Kramer
A general structure for uncooperative processes distributed over a system network.
DISPLAY
14 AUG 73
14 AUG 73
22 JUL 76
10 DEC 76
JAN 5 '78
JAN 12 '78
S10173
23499
23748

Thesis
K848 Kramer
143377
A general structure for uncooperative processes distributed over a system network.

thesK848
A general structure for uncooperative pr
3 2768 002 11515 6
DUDLEY KNOX LIBRARY